

capítulo

2

Complejidad de los algoritmos y cotas inferiores de los problemas

En este capítulo se analizarán algunas cuestiones básicas relacionadas con el análisis de algoritmos. Esencialmente se intentará aclarar los siguientes temas:

1. Algunos algoritmos son eficientes y otros no. ¿Cómo medir la eficiencia de un algoritmo?
2. Algunos problemas son fáciles de resolver y otros no. ¿Cómo medir la dificultad de un problema?
3. ¿Cómo saber si un algoritmo es el óptimo para un problema? Es decir, ¿cómo es posible saber que no existe otro algoritmo mejor para resolver el mismo problema? Mostraremos que todos estos problemas están relacionados entre sí.

2-1 / COMPLEJIDAD TEMPORAL DE UN ALGORITMO

Suele decirse que un algoritmo es bueno si para ejecutarlo se necesita poco tiempo y si requiere poco espacio de memoria. Sin embargo, por tradición, un factor más importante para determinar la eficacia de un algoritmo es el tiempo necesario para ejecutarlo. A lo largo de todo este libro, a menos que se indique lo contrario, el criterio importante es el tiempo.

Para medir la complejidad temporal de un algoritmo, es tentador escribir un programa para este algoritmo y ver qué tan rápido corre. Esto no es lo apropiado porque hay muchos factores no relacionados con el algoritmo que afectan el desempeño del programa. Por ejemplo, la habilidad del programador, el lenguaje usado, el sistema operativo e incluso el compilador del lenguaje particular, todos estos factores afectan el tiempo necesario para ejecutar el programa.

En el análisis de algoritmos siempre se escogerá un tipo de operación particular que ocurra en el algoritmo, y se realizará un análisis matemático a fin de determinar el número de operaciones necesarias para completar el algoritmo. Por ejemplo, en todos

los algoritmos de ordenamiento debe hacerse la comparación de datos, por lo que el número de comparaciones suele emplearse para medir la complejidad temporal de algoritmos de ordenamiento.

Por supuesto, es legítimo refutar que en algunos algoritmos de ordenamiento la comparación de datos no es un factor dominante. De hecho, es fácil poner ejemplos de que en algunos algoritmos de ordenamiento el intercambio de datos es lo que consume más tiempo. Según tal circunstancia, parece que debería usarse el intercambio de datos, y no la comparación, para medir la complejidad temporal de este algoritmo de ordenamiento particular.

Suele decirse que el costo de ejecución de un algoritmo depende del tamaño del problema (n). Por ejemplo, el número de puntos en el problema euclíadiano del agente viajero definido en el apartado 9-2 es el tamaño del problema. Como es de esperar, la mayoría de los algoritmos requiere más tiempo para completar su ejecución a medida que n crece.

Suponga que para ejecutar un algoritmo se requieren $(n^3 + n)$ pasos. Se diría a menudo que la complejidad temporal de este algoritmo es del orden de n^3 . Debido a que el término n^3 es de orden superior a n y a medida que n se hace más grande, el término n pierde importancia en comparación con n^3 . A continuación daremos un significado formal y preciso a esta informal afirmación.

Definición

$f(n) = O(g(n))$ si y sólo si existen dos constantes positivas c y n_0 tales que $|f(n)| \leq c|g(n)|$ para toda $\geq n_0$.

Por la definición anterior se entiende que, si $f(n) = O(g(n))$, entonces $f(n)$ está acotada en cierto sentido por $g(n)$ cuando n es muy grande. Si se afirma que la complejidad temporal de un algoritmo es $O(g(n))$, quiere decir que para ejecutar este algoritmo siempre se requiere de menos que c veces $|g(n)|$ a medida que n es suficientemente grande para alguna c .

A continuación se considerará el caso en que para completar un algoritmo se requieren $(n^3 + n)$ pasos. Así,

$$\begin{aligned} f(n) &= n^3 + n \\ &= \left(1 + \frac{1}{n^2}\right)n^3 \\ &\leq 2n^3 \quad \text{para } n \geq 1. \end{aligned}$$

Por consiguiente, puede afirmarse que la complejidad temporal es $O(n^3)$ porque es posible que c y n_0 sean 2 y 1, respectivamente.

A continuación aclararemos una cuestión muy importante, que suele ser una interpretación equivocada sobre el orden de magnitud de la complejidad temporal de los algoritmos.

Suponga que tiene dos algoritmos A_1 y A_2 que resuelven el mismo problema. Sean $O(n^3)$ y $O(n)$ las complejidades temporales de A_1 y A_2 , respectivamente. Si a una misma persona se le solicita escribir dos programas para A_1 y A_2 y ejecutarlos en el mismo ambiente de programación, ¿el programa para A_2 correrá más rápido que el programa para A_1 ? Un error común es pensar que el programa para A_2 siempre se ejecutará más rápido que el programa para A_1 . En realidad, esto no necesariamente es cierto por una sencilla razón: puede requerirse más tiempo para ejecutar un paso en A_2 que en A_1 . En otras palabras, aunque el número de pasos necesarios para ejecutar A_2 sea menor que los requeridos para A_1 , en algunos casos A_1 se ejecuta más rápido que A_2 . Suponga que cada paso de A_1 tarda 1/100 del tiempo necesario para cada paso de A_2 . Entonces los tiempos de cómputo reales para A_1 y A_2 son n^3 y $100n$, respectivamente. Para $n < 10$, A_1 corre más rápido que A_2 . Para $n > 10$, A_2 corre más rápido que A_1 .

El lector comprende ahora la importancia de la constante que aparece en la definición de la función $O(g(n))$. No es posible ignorarla. Sin embargo, no influye cuán grande sea la constante, su importancia decrece a medida que n crece. Si las complejidades de A_1 y A_2 son $O(g_1(n))$ y $O(g_2(n))$, respectivamente, y $g_1(n) < g_2(n)$ para toda n , se entiende que a medida que n aumenta lo suficiente, A_1 se ejecuta más rápido que A_2 .

Otra cuestión que debe recordarse es que siempre es posible, por lo menos teóricamente, codificar (e implementar) en hardware (hardwire) cualquier algoritmo. Es decir, siempre es posible diseñar un circuito a la medida para implementar un algoritmo. Si dos algoritmos están implementados en hardware, el tiempo necesario para ejecutar un paso de uno de los algoritmos puede igualarse al tiempo necesario que requiere en el otro algoritmo. En tal caso, el orden de magnitud es mucho más importante. Si las complejidades temporales de A_1 y A_2 son $O(n^3)$ y $O(n)$, respectivamente, entonces se sabe que A_2 es mejor que A_1 si ambos están implementados en hardware. Por supuesto, el análisis anterior tiene sentido sólo si se domina a la perfección la habilidad de implementar en hardware los algoritmos.

La importancia del orden de magnitud puede verse al estudiar la tabla 2-1. En esta tabla observamos lo siguiente:

1. Es muy importante si puede encontrarse un algoritmo con menor orden de complejidad temporal. Un caso típico lo constituye la búsqueda. En el peor de los casos, una búsqueda secuencial a través de una lista de n números requiere $O(n)$ operaciones. Si se tiene una lista ordenada de n números, es posible usar búsqueda

TABLA 2-1 Funciones de complejidad temporal.

Función de complejidad temporal	Tamaño del problema (n)			
	10	10^2	10^3	10^4
$\log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n \log_2 n$	0.33×10^2	0.7×10^3	10^4	1.3×10^5
n^2	10^2	10^4	10^6	10^8
2^n	1 024	1.3×10^{30}	$>10^{100}$	$>10^{100}$
$n!$	3×10^6	$>10^{100}$	$>10^{100}$	$>10^{100}$

binaria y la complejidad temporal se reduce a $O(\log_2 n)$, en el peor de los casos. Para $n = 10^4$, la búsqueda secuencial puede requerir 10^4 operaciones, mientras que la búsqueda binaria sólo requiere 14 operaciones.

2. Aunque las funciones de complejidad temporal como n^2 , n^3 , etc., pueden no ser deseables, siguen siendo tolerables en comparación con una función del tipo 2^n . Por ejemplo, cuando $n = 10^4$, entonces $n^2 = 10^8$, pero $2^n > 10^{100}$. El número 10^{100} es tan grande que no importa cuán rápida sea una computadora, no es capaz de resolver este problema. Cualquier algoritmo con complejidad temporal $O(p(n))$, donde $p(n)$ es una función polinomial, es un algoritmo polinomial. Por otra parte, los algoritmos cuyas complejidades temporales no pueden acotarse con una función polinomial son algoritmos exponenciales.

Hay una gran diferencia entre algoritmos polinomiales y algoritmos exponenciales. Lamentablemente, existe una gran clase de algoritmos exponenciales y no parece haber alguna esperanza de que puedan sustituirse por algoritmos polinomiales. Todo algoritmo para resolver el problema eucliano del agente viajero, por ejemplo, es un algoritmo exponencial, hasta ahora. De manera semejante, hasta el presente todo algoritmo para resolver el problema de satisfactibilidad, según se define en la sección 8-3, es un algoritmo exponencial. Aunque como se verá, el problema de árbol de expansión mínima, según se define en el apartado 3-1, puede resolverse con algoritmos polinomiales.

El análisis anterior fue muy vago en cuanto a los datos. Ciertamente, para algunos datos un algoritmo puede terminar bastante rápido, pero para otros datos puede tener un comportamiento completamente distinto. Estos temas se abordarán en el siguiente apartado.

2-2 ANÁLISIS DEL MEJOR CASO, PROMEDIO Y PEOR DE LOS ALGORITMOS

Para cualquier algoritmo, se está interesado en su comportamiento en tres situaciones: el mejor caso, el caso promedio y el peor caso. Por lo general el análisis del mejor caso es el más fácil, el análisis del peor caso es el segundo más fácil y el más difícil es el análisis del caso promedio. De hecho, aún hay muchos problemas abiertos que implican el análisis del caso promedio.

► Ejemplo 2-1 Ordenamiento por inserción directa

Uno de los métodos de ordenamiento más sencillos es el ordenamiento por inserción directa. Se tiene una secuencia de números x_1, x_2, \dots, x_n . Los números se recorren de izquierda a derecha y se escribe x_i a la izquierda de x_{i-1} si x_i es menor que x_{i-1} . En otras palabras, desplazamos a x_i de manera continua hacia la izquierda hasta que los números a su izquierda sean menores que o iguales a él.

Algoritmo 2-1 □ Ordenamiento por inserción directa

```

Input:  $x_1, x_2, \dots, x_n$ .
Output: La secuencia ordenada de  $x_1, x_2, \dots, x_n$ .
    For  $j := 2$  to  $n$  do
        Begin
             $i := j - 1$ 
             $x := x_j$ 
            While  $x < x_i$  and  $i > 0$  do
                Begin
                     $x_{i+1} := x_i$ 
                     $i := i - 1$ 
                End
                 $x_{i+1} := x$ 
        End
    
```

Considere la secuencia de entrada 7, 5, 1, 4, 3, 2, 6. El ordenamiento por inserción directa produce la secuencia ordenada siguiente:

7
 5, 7
 1, 5, 7
 1, 4, 5, 7
 1, 3, 4, 5, 7
 1, 2, 3, 4, 5, 7
 1, 2, 3, 4, 5, 6, 7.

En nuestro análisis, como medida de la complejidad temporal del algoritmo se usará el número de intercambios de datos $x := x_i$, $x_{i+1} := x_i$ y $x_{i+1} := x$. En este algoritmo hay dos ciclos: uno exterior (**for**) y otro interior (**while**). Para el ciclo exterior siempre se ejecutan dos operaciones de intercambio de datos; a saber, $x := x_i$ y $x_{i+1} := x$. Debido a que el ciclo interior puede o no ser ejecutado, el número de intercambios de datos realizados para x_i en el ciclo interior se denotará por d_i . Así, evidentemente, el número total de movimientos de datos para el ordenamiento por inserción directa es

$$\begin{aligned} X &= \sum_{i=2}^n (2 + d_i) \\ &= 2(n - 1) + \sum_{i=2}^n d_i \end{aligned}$$

Mejor caso: $\sum_{i=2}^n d_i = 0$, $X = 2(n - 1) = O(n)$

Esto ocurre cuando los datos de entrada ya están ordenados.

Peor caso: El peor caso ocurre cuando los datos de entrada están ordenados a la inversa. En este caso,

$$\begin{aligned} d_2 &= 1 \\ d_3 &= 2 \\ &\vdots \\ d_n &= n - 1. \end{aligned}$$

De este modo,

$$\sum_{i=2}^n d_i = \frac{n}{2}(n-1)$$

$$X = 2(n-1) + \frac{n}{2}(n-1) = \frac{1}{2}(n-1)(n+4) = O(n^2).$$

Caso promedio: Cuando se está considerando x_i , ya se han ordenado $(i-1)$ datos. Si x_i es el más grande de todos los i números, entonces el ciclo interior no se ejecuta y dentro de este ciclo interior no hay en absoluto ningún movimiento de datos. Si x_i es el segundo más grande de todos los i números, habrá un intercambio de datos, y así sucesivamente. La probabilidad de que x_i sea el más grande es $1/i$. Esta también es la probabilidad de que x_i sea el j -ésimo más grande para $1 \leq j \leq i$. En consecuencia, el promedio $(2 + d_i)$ es

$$\begin{aligned} \frac{2}{i} + \frac{3}{i} + \dots + \frac{i+1}{i} &= \sum_{j=1}^i \frac{(j+1)}{i} \\ &= \frac{i+3}{2}. \end{aligned}$$

La complejidad temporal media para el ordenamiento por inserción directa es

$$\begin{aligned} \sum_{i=2}^n \frac{i+3}{2} &= \frac{1}{2} \left(\sum_{i=2}^n i + \sum_{i=2}^n 3 \right) \\ &= \frac{1}{4}(n-1)(n+8) = O(n^2). \end{aligned}$$

En resumen, la complejidad temporal del ordenamiento por inserción directa para cada caso es:

Mejor caso: $2(n-1) = O(n)$.

Caso promedio: $\frac{1}{4}(n+8)(n-1) = O(n^2)$.

Peor caso: $\frac{1}{2}(n-1)(n+4) = O(n^2)$.

► Ejemplo 2-2 El algoritmo de búsqueda binaria

La búsqueda binaria es un famoso algoritmo de búsqueda. Después de ordenar un conjunto numérico en una secuencia creciente o decreciente, el algoritmo de búsqueda binaria empieza desde la parte media de la secuencia. Si el punto de prueba es igual

al punto medio de la secuencia, finaliza el algoritmo; en caso contrario, dependiendo del resultado de comparar el elemento de prueba y el punto central de la secuencia, de manera recurrente se busca a la izquierda o a la derecha de la secuencia.

Algoritmo 2-2 □ Búsqueda binaria

Input: Un arreglo ordenado $a_1, a_2, \dots, a_n, n > 0$ y X , donde $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$.

Output: j si $a_j = X$ y 0 si no existe ninguna j tal que $a_j = X$.

$i := 1$ (* primer elemento *)

$m := n$ (* último elemento *)

While $i \leq m$ do

Begin

$$j := \left\lfloor \frac{i + m}{2} \right\rfloor$$

If $X = a_j$ then output j y parar

If $X < a_j$ then $m := j - 1$

else $i := j + 1$

End

$j := 0$

Output j

El análisis del mejor caso para la búsqueda binaria es más sencillo. En el mejor caso, la búsqueda binaria termina en un solo paso.

El análisis del peor caso también es bastante sencillo. Resulta fácil ver que para completar la búsqueda binaria se requiere cuando mucho de $(\lceil \log_2 n \rceil + 1)$ pasos. A lo largo de todo este libro, a menos que se indique lo contrario, $\log n$ significa $\log_2 n$.

Para simplificar este análisis, se supondrá que $n = 2^k - 1$.

Para el análisis del caso promedio, se observa que si hay n elementos, entonces hay un elemento para el cual el algoritmo termina exitosamente en un solo paso. Este elemento se localiza en la $\left\lfloor \frac{1+n}{2} \right\rfloor$ -ésima posición de la secuencia ordenada. Hay dos elementos que hacen que la búsqueda binaria termine exitosamente después de dos pasos. En general, hay 2^{t-1} elementos que hacen que la búsqueda binaria termine exitosa-

mente después de t pasos, para $t = 1, 2, \dots, \lfloor \log n \rfloor + 1$. Si X no está en la lista, entonces el algoritmo termina sin éxito después de $\lfloor \log n \rfloor + 1$ pasos. En total, puede decirse que hay $(2n + 1)$ casos distintos: n casos que hacen que la búsqueda termine exitosamente y $(n + 1)$ casos que hacen que la búsqueda termine sin éxito.

Sean $A(n)$ el número medio de comparaciones efectuadas en la búsqueda binaria y $k = \lfloor \log n \rfloor + 1$. Entonces

$$A(n) = \frac{1}{2n+1} \left(\sum_{i=1}^k i2^{i-1} + k(n+1) \right).$$

A continuación se demostrará que

$$\sum_{i=1}^k i2^{i-1} = 2^k(k-1) + 1. \quad (2-1)$$

La fórmula anterior puede demostrarse por inducción sobre k . La ecuación (2-1) es evidentemente cierta para $k = 1$. Suponga que la ecuación (2-1) se cumple para $k = m$, $m > 1$. Entonces se demostrará que es verdadera para $k = m + 1$. Es decir, suponiendo que la ecuación (2-1) es válida, se demostrará que

$$\sum_{i=1}^{m+1} i2^{i-1} = 2^{m+1}(m+1-1) + 1 = 2^{m+1} \cdot m + 1.$$

Observe que

$$\sum_{i=1}^{m+1} i2^{i-1} = \sum_{i=1}^m i2^{i-1} + (m+1)2^{m+1-1}.$$

Al sustituir la ecuación (2-1) en la fórmula anterior se obtiene

$$\begin{aligned} \sum_{i=1}^{m+1} i2^{i-1} &= 2^m(m-1) + 1 + (m+1)2^m \\ &= 2^m \cdot 2m + 1 \\ &= 2^{m+1} \cdot m + 1. \end{aligned}$$

Así, se ha demostrado la validez de la ecuación (2-1). Al usar la ecuación (2-1)

$$A(n) = \frac{1}{2n+1} ((k-1)2^k + 1 + k2^k).$$

Cuando n es muy grande, se tiene

$$\begin{aligned} A(n) &\approx \frac{1}{2^{k+1}} (2^k(k-1) + k2^k) \\ &= \frac{(k-1)}{2} + \frac{k}{2} \\ &= k - \frac{1}{2}. \end{aligned}$$

En consecuencia, $A(n) < k = O(\log n)$.

A hora quizás el lector se pregunte si el resultado obtenido es válido para n en general, habiendo partido de la suposición de que $n = 2^k$. Pensemos que $t(n)$ es una función no decreciente, y $t(n) = O(f(n))$ es la complejidad temporal de nuestro algoritmo, obtenida al suponer que $n = 2^k$ y $f(bn) \leq c'f(n)$ para una $b \geq 1$ y c' es una constante (esto significa que f es una función continua y que toda función polinomial también es así). Entonces

$$t(2^k) \leq cf(2^k) \quad \text{donde } c \text{ es una constante}$$

$$\text{Sea } n' = 2^{k+x} \quad \text{para } 0 \leq x \leq 1$$

$$\begin{aligned} t(n') &= t(2^{k+x}) \\ &\leq t(2^{k+1}) \leq cf(2^{k+1}) \\ &= cf(2^{k+x} \cdot 2^{1-x}) \\ &\leq cc'f(2^{k+x}) = c''f(n'). \end{aligned}$$

En consecuencia, $t(n') = O(f(n'))$.

El análisis anterior muestra que es posible suponer que $n = 2^k$ para obtener la complejidad temporal de un algoritmo. En el resto del libro, siempre que sea necesario, se supondrá que $n = 2^k$ sin explicar por qué es posible hacer esta suposición.

En resumen, para la búsqueda binaria, se tiene

Mejor caso: $O(1)$.

Caso promedio: $O(\log n)$.

Peor caso: $O(\log n)$.

► **Ejemplo 2-3 Ordenamiento por selección directa**

El ordenamiento por selección directa es tal vez el tipo de ordenamiento más sencillo. No obstante, el análisis de este algoritmo es bastante interesante. El ordenamiento por selección directa puede describirse fácilmente como sigue:

1. Se encuentra el número más pequeño. Se hace que este número más pequeño ocupe la posición a_1 mediante el intercambio de a_1 y dicho número.
2. Se repite el paso anterior con los números restantes. Es decir, se encuentra el segundo número más pequeño y se le coloca en a_2 .
3. El proceso continúa hasta que se encuentra el número más grande.

Algoritmo 2-3 □ Ordenamiento por selección directa

Input: a_1, a_2, \dots, a_n .

Output: La secuencia ordenada de a_1, a_2, \dots, a_n .

```

For  $j := 1$  to  $n - 1$  do
  Begin
     $f := j$ 
    For  $k := j + 1$  to  $n$  do
      If  $a_k < a_f$  then  $f := k$ 
     $a_j \leftrightarrow a_f$ 
  End

```

En el algoritmo anterior, para encontrar el número más pequeño de a_1, a_2, \dots, a_n , primeramente se pone una bandera (o pivote) f e inicialmente $f = 1$. Luego a_f se compara con a_2 . Si $a_f < a_2$, no se hace nada; en caso contrario, se hace $f = 2$; a_f se compara con a_3 , y así sucesivamente.

Resulta evidente que en el ordenamiento por selección directa hay dos operaciones: la comparación de dos elementos y el cambio de la bandera. El número de comparaciones entre dos elementos es un número fijo; a saber, $n(n - 1)/2$. Es decir, sin importar cuáles sean los datos de entrada, siempre es necesario efectuar $n(n - 1)/2$ comparaciones. En consecuencia, para medir la complejidad temporal del ordenamiento por selección directa se escogerá el número de cambios de la bandera.

El cambio de la bandera depende de los datos. Considere $n = 2$. Sólo hay dos permutaciones:

$$\begin{array}{ll} (1, 2) \\ \text{y} & (2, 1). \end{array}$$

Para la primera permutación, ningún cambio de la bandera es necesario, mientras que para la segunda se requiere un cambio de la bandera.

Sea $f(a_1, a_2, \dots, a_n)$ que denota el número de cambios de la bandera necesarios para encontrar el número más pequeño para la permutación a_1, a_2, \dots, a_n . La tabla siguiente ilustra el caso para $n = 3$.

$a_1,$	$a_2,$	a_3	$f(a_1, a_2, a_3)$
1,	2,	3	0
1,	3,	2	0
2,	1,	3	1
2,	3,	1	1
3,	1,	2	1
3,	2,	1	2

Para determinar $f(a_1, a_2, \dots, a_n)$ se observa lo siguiente:

- Si $a_n = 1$, entonces $f(a_1, a_2, \dots, a_n) = 1 + f(a_1, a_2, \dots, a_{n-1})$ porque en el último paso debe haber un cambio de la bandera.
- Si $a_n \neq 1$, entonces $f(a_1, a_2, \dots, a_n) = f(a_1, a_2, \dots, a_{n-1})$ porque en el último paso no debe haber ningún cambio de la bandera.

Sea $P_n(k)$ que denota la probabilidad de que una permutación a_1, a_2, \dots, a_n de $\{1, 2, \dots, n\}$ requiera k cambios de la bandera para encontrar el número más pequeño.

Por ejemplo, $P_3(0) = \frac{2}{6}$, $P_3(1) = \frac{3}{6}$ y $P_3(2) = \frac{1}{6}$. Así que, el número promedio de cambios de bandera para encontrar el número más pequeño es

$$X_n = \sum_{k=0}^{n-1} kP_n(k).$$

El número promedio de cambios de bandera para el ordenamiento por selección directa es

$$A(n) = X_n + A(n - 1).$$

Para encontrar X_n se usarán las siguientes ecuaciones, que ya se analizaron:

$$\begin{aligned} f(a_1, a_2, \dots, a_n) &= 1 + f(a_1, a_2, \dots, a_{n-1}) && \text{si } a_n = 1 \\ &= f(a_1, a_2, \dots, a_{n-1}) && \text{si } a_n \neq 1. \end{aligned}$$

Con base en las fórmulas anteriores, se tiene

$$P_n(k) = P(a_n = 1)P_{n-1}(k-1) + P(a_n \neq 1)P_{n-1}(k).$$

Pero $P(a_n = 1) = 1/n$ y $P(a_n \neq 1) = (n-1)/n$. En consecuencia, se tiene

$$P_n(k) = \frac{1}{n}P_{n-1}(k-1) + \frac{n-1}{n}P_{n-1}(k). \quad (2-2)$$

A demás, se tiene la siguiente fórmula, relacionada con las condiciones iniciales:

$$\begin{aligned} P_1(k) &= 1 && \text{si } k = 0 \\ &= 0 && \text{si } k \neq 0 \\ P_n(k) &= 0 && \text{si } k < 0, \text{ y si } k = n. \end{aligned} \quad (2-3)$$

Para proporcionar al lector un conocimiento adicional sobre las fórmulas, se observa que

$$\begin{aligned} P_2(0) &= \frac{1}{2} \\ \text{y } P_2(1) &= \frac{1}{2}; \\ P_3(0) &= \frac{1}{3}P_2(-1) + \frac{2}{3}P_2(0) = \frac{1}{3} \times 0 + \frac{2}{3} \times \frac{1}{2} = \frac{1}{3} \\ \text{y } P_3(2) &= \frac{1}{3}P_2(1) + \frac{2}{3}P_2(2) = \frac{1}{3} \times \frac{1}{2} + \frac{2}{3} \times 0 = \frac{1}{6}. \end{aligned}$$

A continuación, se demostrará que

$$\chi_n = \sum_{k=1}^{n-1} kP_n(k) = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} = H_n - 1, \quad (2-4)$$

donde H_n es el n -ésimo número armónico.

La ecuación (2-4) puede demostrarse por inducción. Es decir, se quiere demostrar que

$$\chi_n = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} = H_n - 1$$

es trivialmente cierta para $n = 2$, y suponiendo que la ecuación (2-4) se cumple cuando $n = m$ para $m > 2$, se demostrará que la ecuación (2-4) es verdadera para $n = m + 1$. Así, se quiere demostrar que

$$\begin{aligned} X_{m+1} &= H_{m+1} - 1 \\ X_{m+1} &= \sum_{k=1}^m kP_{m+1}(k) \\ &= P_{m+1}(1) + 2P_{m+1}(2) + \dots + mP_{m+1}(m). \end{aligned}$$

Al usar la ecuación (2-2) se tiene

$$\begin{aligned} X_{m+1} &= \frac{1}{m+1} P_m(0) + \frac{m}{m+1} P_m(1) + \frac{2}{m+1} P_m(1) + \frac{2m}{m+1} P_m(2) \\ &\quad + \dots + \frac{m}{m+1} P_m(m-1) + \frac{m^2}{m+1} P_m(m) \\ &= \frac{1}{m+1} P_m(0) + \frac{m}{m+1} P_m(1) + \frac{1}{m+1} P_m(1) + \frac{1}{m+1} P_m(1) \\ &\quad + \frac{2m}{m+1} P_m(2) + \frac{1}{m+1} P_m(2) + \frac{2}{m+1} P_m(2) \\ &\quad + \frac{3m}{m+1} P_m(3) + \dots + \frac{1}{m+1} P_m(m-1) + \frac{m-1}{m+1} P_m(m-1) \\ &= \frac{1}{m+1} (P_m(0) + P_m(1) + \dots + P_m(m-1)) + \frac{m+1}{m+1} P_m(1) \\ &\quad + \frac{2m+2}{m+1} P_m(2) + \dots + \frac{(m-1)(m+1)}{m+1} P_m(m-1) \\ &= \frac{1}{m+1} + (P_m(1) + 2P_m(2) + \dots + (m-1)P_m(m-1)) \\ &= \frac{1}{m+1} + H_m - 1 \quad (\text{por hipótesis de inducción}) \\ &= H_{m+1} - 1. \end{aligned}$$

Debido a que la complejidad temporal del ordenamiento por selección directa es

$$A(n) = X_n + A(n-1),$$

se tiene

$$\begin{aligned}
 A(n) &= H_n - 1 + A(n-1) \\
 &= (H_n - 1) + (H_{n-1} - 1) + \cdots + (H_2 - 1) \\
 &= \sum_{i=2}^n H_i - (n-1) \\
 \sum_{i=1}^n H_i &= H_n + H_{n-1} + \cdots + H_1 \\
 &= H_n + \left(H_n - \frac{1}{n} \right) + \cdots + \left(H_n - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{2} \right) \\
 &= nH_n - \left(\frac{n-1}{n} + \frac{n-2}{n-1} + \cdots + \frac{1}{2} \right) \\
 &= nH_n - \left(1 - \frac{1}{n} + 1 - \frac{1}{n-1} + \cdots + 1 - \frac{1}{2} \right) \\
 &= nH_n - \left(n - 1 - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{2} \right) \\
 &= nH_n - n + H_n \\
 &= (n+1)H_n - n.
 \end{aligned} \tag{2-5}$$

En consecuencia,

$$\sum_{i=2}^n H_i = (n+1)H_n - H_1 - n. \tag{2-6}$$

Al sustituir la ecuación (2-6) en la ecuación (2-5) se tiene

$$\begin{aligned}
 A(n) &= (n+1)H_n - H_1 - (n-1) - n \\
 &= (n+1)H_n - 2n.
 \end{aligned}$$

A medida que n es suficientemente grande,

$$A(n) \cong n \log_e n = O(n \log n).$$

Las complejidades temporales del ordenamiento por selección directa pueden resumirse como sigue:

Mejor caso: $O(1)$.

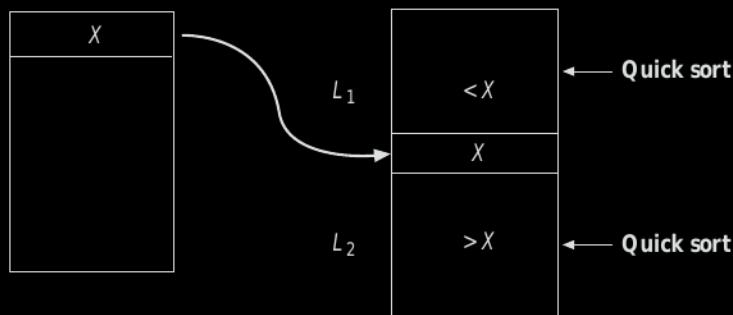
Peor caso: $O(n \log n)$.

Caso promedio: $O(n^2)$.

► Ejemplo 2-4 El algoritmo quick sort

El algoritmo quick sort se basa en la estrategia divide-y-vencerás (divide-and-conquer), que se ilustrará con todo detalle en otro momento. Por ahora, es posible afirmar que esta estrategia divide un problema en dos subproblemas, que se resuelven de manera individual e independiente. Los resultados se unen después. Al aplicar la estrategia divide-y-vencerás al ordenamiento se obtiene un método para clasificar denominado quick sort. Dado un conjunto de números a_1, a_2, \dots, a_n , se escoge un elemento X para dividir dicho conjunto en dos listas, como se muestra en la figura 2-1.

FIGURA 2-1 Quick sort.



Después de la división, el quick sort puede aplicarse en forma recursiva tanto a L_1 como a L_2 , con lo cual se obtiene una lista ordenada, ya que L_1 contiene a todos los a_i menores que o iguales a X y L_2 contiene a todos los a_i mayores que X .

El problema es: ¿cómo dividir la lista original? Ciertamente, no debe usarse X para recorrer toda la lista y decidir si un a_i es menor que o igual a X . Hacer lo anterior provoca una gran cantidad de intercambio de datos. Como se muestra en nuestro algoritmo, se utilizan dos apuntadores que se mueven al centro y realizan intercambios de datos según sea necesario.

Algoritmo 2-4 □ Quick sort (f, l)

Input: a_f, a_{f+1}, \dots, a_l .

Output: La secuencia ordenada de a_f, a_{f+1}, \dots, a_l .

If $f \geq l$ then Return

$X := a_f$

$i := f + 1$

$j := l$

While $i < j$ do

Begin

While $a_j \geq X$ y $j \geq f + 1$ do

$j := j - 1$

While $a_i \leq X$ y $i \leq f + l$ do

$i := i + 1$

Si $i < j$ entonces $a_i \leftrightarrow a_j$

End

Quick sort($f, j - 1$)

Quick sort($j + 1, l$)

El siguiente ejemplo ilustra la característica más importante del algoritmo quick sort:

$X = 3$	a_1	a_2	a_3	a_4	a_5	a_6
$a_j = a_6 < X$	3	6	1	4	5	2
	$\uparrow i$					$\uparrow j$
	2	6	1	4	5	3
	$\uparrow i$					$\uparrow j$
$a_i = a_2 > X$	2	6	1	4	5	3
	$\uparrow i$					$\uparrow j$
	2	3	1	4	5	6
	$\uparrow i$					$\uparrow j$
	2	3	1	4	5	6
	$\uparrow i$					$\uparrow j$
	2	3	1	4	5	6
	$\uparrow i$					$\uparrow j$

$a_j = a_3 < X$	2	3	1	4	5	6
	$\uparrow i$	$\uparrow j$				
	2	1	3	4	5	6
		$\uparrow i$	$\uparrow j$			
	2	1	3	4	5	6
			$i \uparrow \uparrow j$			
	$ \leftarrow \leq 3 \rightarrow $	=3	$ \leftarrow \geq 3 \rightarrow $			

El mejor caso de quick sort ocurre cuando X divide la lista justo en el centro. Es decir, X produce dos sublistas que contienen el mismo número de elementos. En este caso, la primera ronda requiere n pasos para dividir la lista original en dos listas. Para la ronda siguiente, para cada sublista, de nuevo se necesitan $n/2$ pasos (ignorando el elemento usado para la división). En consecuencia, para la segunda ronda nuevamente se requieren $2 \cdot (n/2) = n$ pasos. Si se supone que $n = 2^p$, entonces en total se requieren pn pasos. Sin embargo, $p = \log_2 n$. Así, para el mejor caso, la complejidad temporal del quick sort es $O(n \log n)$.

El peor caso del quick sort ocurre cuando los datos de entrada están ya ordenados o inversamente ordenados. En estos casos, todo el tiempo simplemente se está seleccionando el extremo (ya sea el mayor o el menor). Por lo tanto, el número total de pasos que se requiere en el quick sort para el peor caso es

$$n + (n - 1) + \dots + 1 = \frac{n}{2}(n + 1) = O(n^2).$$

Para analizar el caso promedio, sea $T(n)$ que denota el número de pasos necesarios para llevar a cabo el quick sort en el caso promedio para n elementos. Se supondrá que después de la operación de división la lista se ha dividido en dos sublistas. La primera de ellas contiene s elementos y la segunda contiene $(n - s)$ elementos. El valor de s varía desde 1 hasta n y es necesario tomar en consideración todos los casos posibles a fin de obtener el desempeño del caso promedio. Para obtener $T(n)$ es posible aplicar la siguiente fórmula:

$$T(n) = \underset{1 \leq s \leq n}{\text{Promedio}}(T(s) + T(n - s)) + cn$$

donde cn denota el número de operaciones necesario para efectuar la primera operación de división. (Observe que cada elemento es analizado antes de dividir en dos sublistas la lista original.)

$$\begin{aligned}
 & \underset{1 \leq s \leq n}{\text{Promedio}}(T(s) + T(n - s)) \\
 &= \frac{1}{n} \sum_{s=1}^n (T(s) + T(n - s)) \\
 &= \frac{1}{n} (T(1) + T(n - 1) + T(2) + T(n - 2) + \dots + T(n) + T(0)).
 \end{aligned}$$

Debido a que $T(0) = 0$,

$$\begin{aligned}
 T(n) &= \frac{1}{n} (2T(1) + 2T(2) + \dots + 2T(n - 1) + T(n)) + cn \\
 \text{o, } (n - 1)T(n) &= 2T(1) + 2T(2) + \dots + 2T(n - 1) + cn^2.
 \end{aligned}$$

A l sustituir $n = n - 1$ en la fórmula anterior, se tiene

$$(n - 2)T(n - 1) = 2T(1) + 2T(2) + \dots + 2T(n - 2) + c(n - 1)^2.$$

En consecuencia,

$$(n - 1)T(n) - (n - 2)T(n - 1) = 2T(n - 1) + c(2n - 1)$$

$$(n - 1)T(n) - nT(n - 1) = c(2n - 1)$$

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + c\left(\frac{1}{n} + \frac{1}{n-1}\right).$$

En forma recursiva,

$$\begin{aligned}
 \frac{T(n-1)}{n-1} &= \frac{T(n-2)}{n-2} + c\left(\frac{1}{n-1} + \frac{1}{n-2}\right) \\
 &\vdots
 \end{aligned}$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c\left(\frac{1}{2} + \frac{1}{1}\right).$$

Se tiene

$$\begin{aligned}
 \frac{T(n)}{n} &= c\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2} + \cdots + 1\right) \\
 &= c(H_n - 1) + cH_{n-1} \\
 &= c(H_n + H_{n-1} - 1) \\
 &= c\left(2H_n - \frac{1}{n} - 1\right) \\
 &= c\left(2H_n - \frac{n+1}{n}\right).
 \end{aligned}$$

Finalmente, se tiene

$$\begin{aligned}
 T(n) &= 2cnH_n - c(n+1) \\
 &\approx 2cn \log_e n - c(n+1) \\
 &= O(n \log n).
 \end{aligned}$$

En conclusión, las complejidades temporales para el quick sort son

Mejor caso: $O(n \log n)$.

Caso promedio: $O(n \log n)$.

Peor caso: $O(n^2)$.

► Ejemplo 2-5 El problema de encontrar rangos

Suponga que se tiene un conjunto de números a_1, a_2, \dots, a_n . Se dice que a_i domina a a_j si $a_i > a_j$. Si se quiere determinar la cantidad de a_j dominados por un número a_i , entonces simplemente estos números pueden ordenarse en una secuencia, con lo cual el problema se resuelve de inmediato.

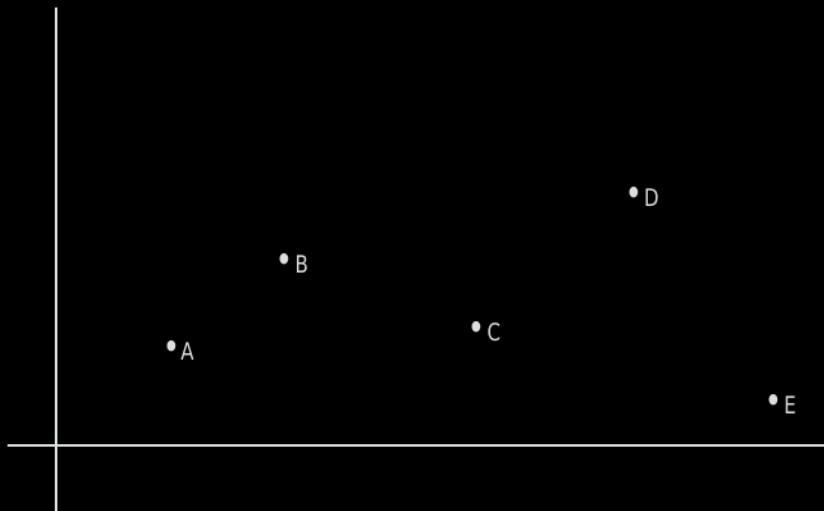
A continuación, este problema se extiende al caso de dos dimensiones. Es decir, cada dato es un punto en el plano. En este caso, primero se define lo que se entiende por dominancia de puntos en el espacio bidimensional.

Definición

Dados dos puntos $A = (a_1, a_2)$ y $B = (b_1, b_2)$, A domina a B si y sólo si $a_i > b_i$ para $i = 1, 2$. Si ocurre que A no domina a B ni B domina a A , entonces A y B no pueden compararse.

Considere la figura 2-2.

FIGURA 2-2 Un caso para ilustrar la relación de dominancia.



Para los puntos de la figura 2-2 se tiene la siguiente relación:

1. B, C y D dominan a A .
2. D domina a A, B y C .
3. No es posible comparar ninguno de los demás pares de puntos.

Una vez que se ha definido la relación de dominancia, es posible definir el rango de un punto.

Definición

Dado un conjunto S de n puntos, el rango de un punto X es el número de puntos dominados por X .

Para los puntos en la figura 2-2, los rangos de A y E son cero porque no dominan a ningún punto. Los rangos de B y C son uno porque A , y sólo A , es domina-

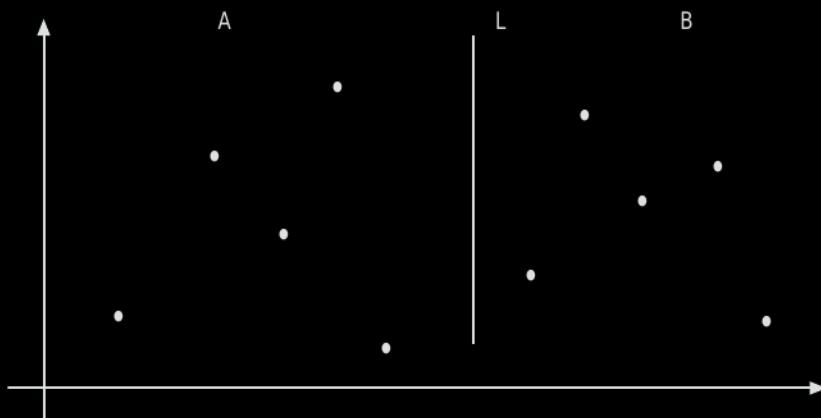
do por ellos. El rango de D es tres porque los tres puntos A , B y C son dominados por D .

El problema consiste en encontrar el rango de cada punto.

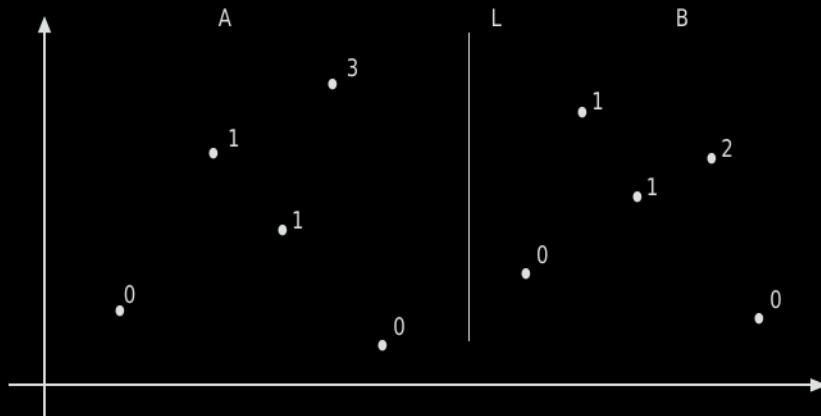
Una forma directa de resolver este problema es realizar una comparación exhaustiva de todos los pares de puntos. La complejidad temporal de este método es $O(n^2)$. A continuación se demostrará que este problema puede resolverse en $O(n \log_2 n^2)$ en el peor caso.

Considere la figura 2-3. El primer paso consiste en encontrar una línea recta L perpendicular al eje x que separe al conjunto de puntos en dos subconjuntos del mismo tamaño. Sean A , que denota el subconjunto a la izquierda de L , y B , que denota el subconjunto a la derecha de L . Resulta evidente que el rango de cualquier punto en A no es afectado por la presencia de B . Sin embargo, el rango de un punto en B puede ser afectado por la presencia de A .

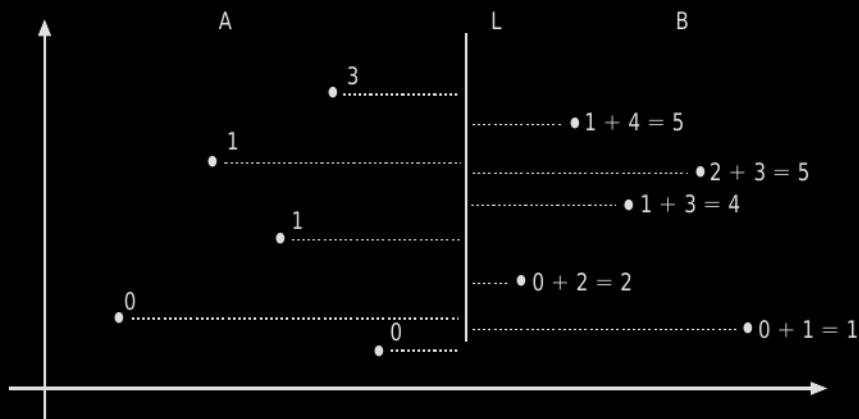
FIGURA 2-3 El primer paso para resolver el problema de determinación del rango.



Supongamos que encontramos los rangos locales de los puntos de A y B por separado. Es decir, encontramos los rangos de los puntos en A (sin tener en cuenta a B) y los rangos de los puntos en B (sin considerar a A). A continuación representamos los rangos locales de puntos de la figura 2-3 en la figura 2-4:

FIGURA 2-4 Los rangos locales de los puntos A y B.

Luego se proyectan todos los puntos sobre L . Observe que un punto P_1 en B domina a un punto P_2 en A si y sólo si el valor y de P_1 es mayor que el de P_2 . Sea P un punto en B . El rango de P es el rango de P , entre los puntos en B , más el número de puntos en A cuyos valores y son más pequeños que el valor y de P . Esta modificación se ilustra en la figura 2-5.

FIGURA 2-5 Modificación de rangos.

El siguiente algoritmo determina el rango de cualquier punto en un plano.

Algoritmo 2-5 □ Algoritmo para encontrar un rango

Input: Un conjunto S de puntos en el plano P_1, P_2, \dots, P_n .

Output: El rango de todos los puntos en S .

- Paso 1.** Si S contiene un solo punto, devuelve su rango como 0. En caso contrario, escoge una línea de corte L perpendicular al eje x de modo que el valor X de $n/2$ puntos de S sea menor que L (este conjunto de puntos se denomina A) y que el valor X de los demás puntos sea mayor que L (este conjunto de puntos se denomina B). Observe que L es el valor X mediana (medida central estadística) de este conjunto.
- Paso 2.** En forma recurrente, este algoritmo para encontrar un rango se usa para encontrar los rangos de los puntos en A y los rangos de los puntos en B .
- Paso 3.** Los puntos en A y en B se clasifican según sus valores y . Estos puntos se analizan secuencialmente y se determina, para cada punto en B , el número de puntos en A cuyos valores y son menores que su valor y . El rango de este punto es igual al rango de este punto entre los puntos en B (lo cual se determinó en el paso 2), más el número de puntos en A cuyos valores y son menores que su valor y .

El algoritmo 2-5 es recursivo. Se basa en la estrategia divide-y-vencerás, que separa un problema en dos subproblemas, resuelve de manera independiente estos dos subproblemas y después une las subsoluciones en la solución final. La complejidad temporal de este algoritmo depende de las complejidades temporales de los pasos siguientes:

1. En el paso 1 hay una operación con la que se encuentra la mediana de un conjunto de números. Más adelante, en el capítulo 7, se demostrará que la menor complejidad temporal de cualquier algoritmo para encontrar la mediana es $O(n)$.
2. En el paso 3 hay una operación de ordenamiento. En la sección 2-3 de este capítulo se demostrará que la menor complejidad temporal de cualquier algoritmo de ordenamiento es $O(n \log n)$. La inspección se lleva a cabo en $O(n)$ pasos.

Sea $T(n)$ que denota el tiempo total necesario para completar el algoritmo para encontrar rangos. Entonces

$$\begin{aligned} T(n) &\leq 2T(n/2) + c_1n \log n + c_2n \\ &\leq 2T(n/2) + cn \log n \quad \text{donde } c_1, c_2 \text{ y } c \text{ son constantes para un } n \text{ suficientemente grande.} \end{aligned}$$

Sea $n = 2^p$. Entonces

$$\begin{aligned}
 T(n) &\leq 2T(n/2) + cn \log n \\
 &\leq 2(2T(n/4) + cn \log (n/2)/2) + cn \log n \\
 &= 4T(n/4) + c(n \log (n/2) + n \log n) \\
 &\quad \vdots \\
 &\leq nT(1) + c(n \log n + n \log (n/2) + n \log (n/4) + \dots + n \log 2) \\
 &= nT(1) + cn\left(\frac{1+\log n}{2} \log n\right) \\
 &= c_3n + \frac{c}{2}n \log^2 n + \frac{c}{2}n \log n.
 \end{aligned}$$

En consecuencia, $T(n) = O(n \log^2 n)$.

La complejidad temporal anterior es para el peor caso y para el caso promedio, ya que la complejidad temporal mínima $O(n \log n)$ para clasificar también es válida para el caso promedio. De manera semejante, la complejidad temporal de $O(n)$ para encontrar la mediana también es válida para el caso promedio.

Observe que este algoritmo para encontrar el rango de cada punto es mucho mejor que un algoritmo en que se utilice una búsqueda exhaustiva. Si ésta se lleva a cabo en todos los pares, entonces para completar el proceso se requieren $O(n^2)$ pasos.

2-3 LA COTA INFERIOR DE UN PROBLEMA

En el apartado anterior, muchos ejemplos de algoritmos para encontrar complejidades temporales nos enseñaron la forma de determinar la eficiencia de un algoritmo. En esta sección, el problema de la complejidad temporal se abordará desde un punto de vista completamente distinto.

Considere el problema de encontrar rangos, o el problema del agente viajero. A hora se pregunta: ¿cómo medir la dificultad de un problema?

Esta pregunta puede responderse de una forma muy intuitiva. Si un problema puede resolverse con un algoritmo con baja complejidad temporal, entonces el problema es sencillo; en caso contrario, se trata de un problema difícil. Esta definición intuitiva conduce al concepto de cota inferior de un problema.

Definición

La cota inferior de un problema es la complejidad temporal mínima requerida por cualquier algoritmo que pueda aplicarse para resolverlo.

Para describir la cota inferior se usará la notación Ω , que se define como sigue:

Definición

$f(n) = \Omega(g(n))$ si y sólo si existen constantes positivas c y n_0 tales que para toda $n > n_0$, $|f(n)| \geq c|g(n)|$.

La complejidad temporal usada en la definición anterior se refiere a la complejidad temporal del peor caso, aunque también puede usarse la complejidad temporal del caso promedio. Si se utiliza la complejidad temporal del caso promedio, la cota inferior se denomina cota inferior del caso promedio; en caso contrario, se denomina cota inferior del peor caso. En todo este libro, a menos que se indique otra cosa, cuando se mencione una cota inferior se entiende que se trata de la cota inferior del peor caso.

Por la definición, casi parece necesario enumerar todos los algoritmos posibles, determinar la complejidad temporal de cada algoritmo y encontrar la complejidad temporal mínima. Por supuesto, lo anterior es utópico porque sencillamente no pueden enumerarse todos los algoritmos posibles; nunca se tiene la certeza de que no vaya a inventarse un nuevo algoritmo que pueda producir una mejor cota inferior.

El lector debe observar que una cota inferior, por definición, no es única. Una cota inferior famosa es la cota inferior para el problema de ordenamiento, que es igual a $\Omega(n \log n)$. Imagine que antes de encontrar esta cota inferior, alguien puede demostrar que una cota inferior para el ordenamiento es $\Omega(n)$ porque cada dato debe examinarse antes de terminar el ordenamiento. De hecho, es posible ser más radical. Por ejemplo, antes de sugerir $\Omega(n)$ como una cota inferior para ordenar, se podría sugerir $\Omega(1)$ como la cota inferior porque para terminar todo algoritmo realiza al menos un paso.

Con base en el análisis anterior, se sabe que puede haber tres cotas inferiores: $\Omega(n \log n)$, $\Omega(n)$ y $\Omega(1)$ para ordenar. Todas son correctas aunque, evidentemente, la única importante es $\Omega(n \log n)$. Las otras dos son cotas inferiores triviales. Debido a que una cota inferior es trivial si es demasiado baja, es deseable que la cota inferior sea lo más alta posible. La búsqueda de cotas inferiores siempre parte de una cota inferior bastante baja sugerida por un investigador. Luego, alguien mejoraría esta cota inferior al encontrar una cota inferior más alta. Esta cota inferior más alta sustituye a la anterior y se convierte en la cota inferior significativa de este problema. Esta situación prevalece hasta que se encuentra una cota inferior aún más alta.

Es necesario entender que cada cota inferior más alta se encuentra mediante un análisis teórico, no por pura suposición. A medida que la cota inferior aumenta, inevitablemente surge la pregunta de si existe algún límite para la cota inferior. Consideré, por ejemplo, la cota inferior para ordenar. ¿Hay alguna posibilidad de sugerir que $\Omega(n^2)$ sea una cota inferior para clasificar? No, porque hay un algoritmo para ordenar,

digamos el algoritmo de ordenamiento heap sort, cuya complejidad temporal en el peor caso es $O(n \log n)$. En consecuencia, se sabe que la cota inferior para clasificar es a lo sumo $\Omega(n \log n)$.

A continuación se considerarán los dos casos siguientes:

Caso 1. En el presente se encuentra que la máxima cota inferior de un problema es $\Omega(n \log n)$ y que la complejidad temporal del mejor algoritmo disponible para resolver este problema es $O(n^2)$.

En este caso hay tres posibilidades:

1. La cota inferior del problema es demasiado baja. En consecuencia, es necesario tratar de encontrar una cota inferior más precisa, o alta. En otras palabras, es necesario intentar mover hacia arriba la cota inferior.
2. El mejor algoritmo disponible no es suficientemente bueno. Por lo tanto, es necesario tratar de encontrar un mejor algoritmo cuya complejidad temporal sea más baja. Dicho de otra manera, es necesario intentar mover hacia abajo la mejor complejidad temporal.
3. La cota inferior puede mejorarse y también es posible mejorar el algoritmo. Por consiguiente, es necesario tratar de mejorar ambos.

Caso 2. La cota inferior actual es $\Omega(n \log n)$ y se requiere un algoritmo cuya complejidad temporal sea $O(n \log n)$.

En este caso se dice que ya no es posible mejorar más ni la cota inferior ni el algoritmo. En otras palabras, se ha encontrado un algoritmo óptimo para resolver el problema, así como una cota inferior verdaderamente significativa para el problema.

A continuación se recalcará la cuestión anterior. Observe que desde el inicio de este capítulo se planteó una pregunta: ¿cómo se sabe que un algoritmo es el óptimo? A hora ya se tiene la respuesta. *Un algoritmo es el óptimo si su complejidad temporal es igual a una cota inferior de este problema. Ya no es posible mejorar más ni la cota inferior ni el algoritmo.*

A continuación se presenta un resumen de algunos conceptos importantes concernientes a las cotas inferiores:

1. La cota inferior de un problema es la complejidad temporal mínima de todos los algoritmos que puede aplicarse para resolverlo.
2. La mejor cota inferior es la más alta.
3. Si la cota inferior conocida actual de un problema es más baja que la complejidad temporal del mejor algoritmo disponible para resolver este problema, entonces es posible mejorar la cota inferior moviéndola hacia arriba. El algoritmo puede mejorarse moviendo hacia abajo su complejidad temporal, o ambas cosas.

4. Si la cota inferior conocida actual es igual a la complejidad temporal de un algoritmo disponible, entonces ya no es posible mejorar más ni el algoritmo ni la cota inferior. El algoritmo es un algoritmo óptimo y la cota inferior es la máxima cota inferior, que es la verdaderamente importante.

En las siguientes secciones se analizarán algunos métodos para encontrar cotas inferiores.

2-4 / LA COTA INFERIOR DEL PEOR CASO DEL ORDENAMIENTO

Para muchos algoritmos, la ejecución del algoritmo puede describirse con árboles binarios. Por ejemplo, considere el caso del ordenamiento por inserción directa. Se supondrá que el número de entradas es tres y que todos los datos son diferentes. En este caso hay seis permutaciones distintas:

a₁	a₂	a₃
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Una vez que se aplica el ordenamiento por inserción directa al conjunto de datos anteriores, cada permutación evoca una respuesta distinta. Por ejemplo, suponga que la entrada es $(2, 3, 1)$. El ordenamiento por inserción directa se comporta ahora como sigue:

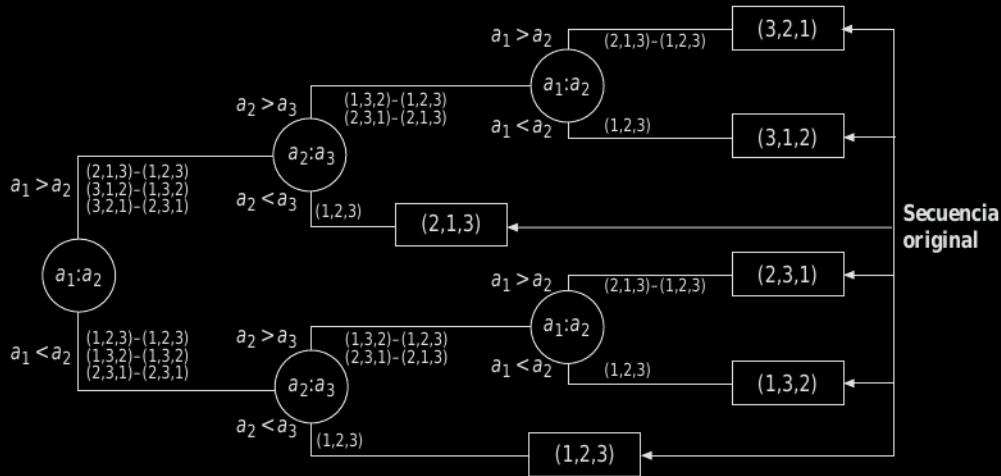
1. $a_1 = 2$ se compara con $a_2 = 3$. Debido a que $a_2 > a_1$, no se realiza ningún cambio de datos.
2. $a_2 = 3$ se compara con $a_3 = 1$. Debido a que $a_2 > a_3$, se intercambian a_2 y a_3 . Es decir, $a_2 = 1$ y $a_3 = 3$.
3. $a_1 = 2$ se compara con $a_3 = 1$. Debido a que $a_1 > a_2$, se intercambian a_1 y a_2 . Es decir, $a_1 = 1$ y $a_2 = 2$.

Si los datos de entrada son $(2, 1, 3)$, entonces el algoritmo se comporta como sigue:

1. $a_1 = 2$ se compara con $a_2 = 1$. Debido a que $a_1 > a_2$, se intercambian a_1 y a_2 . Es decir, $a_1 = 1$ y $a_2 = 2$.
2. $a_2 = 2$ se compara con $a_3 = 3$. Debido a que $a_2 < a_3$, no se realiza ningún cambio de datos.

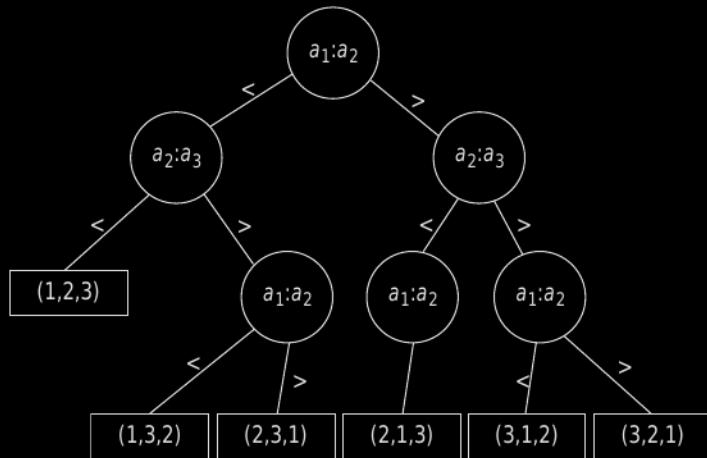
En la figura 2-6 se muestra la forma en que es posible describir el ordenamiento por inserción directa mediante un árbol binario cuando se clasifican tres datos. Este árbol binario puede modificarse fácilmente para manipular el caso en que hay cuatro datos. Es fácil ver que el ordenamiento por inserción directa, cuando se aplica a cualquier número de datos, puede describirse mediante un árbol de decisión binario.

FIGURA 2-6 Ordenamiento por inserción directa con tres elementos representados por un árbol.



En general, cualquier algoritmo de ordenamiento cuya operación básica sea una operación de comparación e intercambio puede describirse mediante un árbol de decisión binario. En la figura 2-7 se muestra cómo es posible describir el ordenamiento por el método de la burbuja (bubble sort) mediante un árbol de decisión binario. En la figura 2-7 se supone que hay tres datos distintos entre sí. Esta figura también está bastante simplificada para que no ocupe mucho espacio. Aquí no se analizará el ordenamiento por el método de la burbuja, ya que es bastante conocido.

FIGURA 2-7 Árbol de decisión binaria que describe el ordenamiento por el método de la burbuja.



La acción de un algoritmo de ordenamiento basado en operaciones de comparación e intercambio sobre un conjunto de datos de entrada particular corresponde a una ruta que va de la raíz del árbol a un nodo hoja. En consecuencia, cada nodo hoja corresponde a una permutación particular. *La ruta más larga que va de la raíz del árbol a un nodo hoja, que se denomina profundidad (altura) del árbol, representa la complejidad temporal del peor caso de este algoritmo.* Para encontrar la cota inferior del problema de ordenamiento es necesario encontrar la profundidad más pequeña de algún árbol de entre todos los algoritmos de ordenamiento de modelado de árboles de decisión binarios posibles.

A continuación se mencionan algunas cuestiones importantes:

1. Para todo algoritmo de ordenamiento, su árbol de decisión binario correspondiente tendrá $n!$ nodos hoja a medida que haya $n!$ permutaciones distintas.
2. La profundidad de un árbol binario con un número fijo de nodos hoja será mínima si el árbol está balanceado.
3. Cuando un árbol binario está balanceado, la profundidad del árbol es $\lceil \log n! \rceil$, donde X es el número de nodos hoja.

A partir del razonamiento anterior, es posible concluir fácilmente que *una cota inferior del problema de ordenamiento es $\lceil \log n! \rceil$* . Es decir, el número de comparaciones necesarias para ordenar en el peor caso es por lo menos $\lceil \log n! \rceil$.

Quizás este $\log n!$ sea algo misterioso para muchos. Simplemente se ignora cuán grande es este número; a continuación se analizarán dos métodos de aproximación a $\log n!$

Método 1.

Se usa el hecho de que

$$\begin{aligned}
 \log n! &= \log(n(n - 1) \dots 1) \\
 &= \sum_{i=1}^n \log i \\
 &= (2 - 1) \log 2 + (3 - 2) \log 3 + \dots + (n - n + 1) \log n \\
 &> \int_1^n \log x dx \\
 &= \log e \int_1^n \log_e x dx \\
 &= \log e[x \log_e x - x]_1^n \\
 &= \log e(n \log_e n - n + 1) \\
 &= n \log n - n \log e + 1.44 \\
 &\geq n \log n - 1.44n \\
 &= n \log n \left(1 - \frac{1.44}{\log n}\right).
 \end{aligned}$$

Si se hace $n = 2^2$, $n \log n \left(1 - \frac{1.44}{2}\right) = 0.28n \log n$.

A sí, al hacer $n_0 = 2^2$ y $c = 0.28$, se tiene

$$\log n! \geq cn \log n \quad \text{para } n \geq n_0.$$

Es decir, una cota inferior para el peor caso de ordenamiento es $\Omega(n \log n)$.

Método 2. Aproximación de Stirling

Con la aproximación de Stirling se aproxima el valor $n!$ a medida que n es muy grande por medio de la siguiente fórmula:

$$n! \cong \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Esta fórmula puede encontrarse en casi cualquier libro de cálculo avanzado. La tabla 2-2 ilustra qué tan bien la aproximación de Stirling se acerca a $n!$ En la tabla, S_n será la aproximación de Stirling.

TABLA 2-2 Algunos valores de la aproximación de Stirling.

<i>n</i>	<i>n!</i>	<i>S_n</i>
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3 628 800	3 598 600
20	2.433×10^{18}	2.423×10^{18}
100	9.333×10^{157}	9.328×10^{157}

Al usar la aproximación de Stirling se tiene

$$\begin{aligned}\log n! &= \log \sqrt{2\pi} + \frac{1}{2} \log n + n \log \frac{n}{e} \\ &= \log \sqrt{2\pi} + \frac{1}{2} \log n + n \log n - n \log e \\ &\geq n \log n - 1.44n.\end{aligned}$$

Con base en ambos métodos es posible afirmar que el número mínimo de comparaciones requerido por el ordenamiento es $\Omega(n \log n)$, en el peor caso.

En este instante es necesario observar que la afirmación anterior no significa que no es posible encontrar una cota inferior. En otras palabras, es posible que algún descubrimiento nuevo pudiera dar a conocer que la cota inferior del ordenamiento es en realidad más alta. Por ejemplo, es posible que alguien pudiera descubrir que la cota inferior del ordenamiento fuera $\Omega(n^2)$.

En la siguiente sección se presentará un algoritmo de ordenamiento cuya complejidad temporal para el peor caso es igual a la cota inferior que acaba de deducirse. Debido a la existencia de tal algoritmo, ya no es posible hacer más alta esta cota inferior.

2-5 / ORDENAMIENTO HEAP SORT: UN ALGORITMO

DE ORDENAMIENTO ÓPTIMO EN EL PEOR CASO

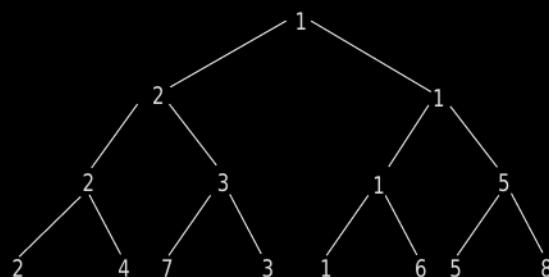
El ordenamiento heap sort forma parte de los algoritmos de ordenamiento cuyas complejidades temporales son $O(n \log n)$. Antes de presentar este tipo de ordenamiento, se analizará el ordenamiento por selección directa a fin de ver por qué no es óptimo en el peor caso. En el ordenamiento por selección directa se requieren $(n - 1)$ pasos para obtener el número más pequeño, y luego, $(n - 2)$ pasos para obtener el segundo núme-

ro más pequeño, y así sucesivamente (siempre en los peores casos). Por consiguiente, en el peor caso, para el ordenamiento por selección directa se requieren $O(n^2)$ pasos. Si el ordenamiento por selección directa se analiza con mayor detenimiento, se observa que cuando se trata de encontrar el segundo número más pequeño, la información que se obtuvo al encontrar el primer número más pequeño no se usa en absoluto. Por eso el ordenamiento por selección directa se comporta de manera tan torpe.

A continuación se considerará otro algoritmo de ordenamiento, denominado knockout sort, que es mucho mejor que el ordenamiento por selección directa. Este ordenamiento es semejante al ordenamiento por selección directa en el sentido de que encuentra el número más pequeño, el segundo más pequeño, y así sucesivamente. No obstante, mantiene cierta información después de encontrar el primer número más pequeño, por lo que es bastante eficiente para encontrar el segundo número más pequeño.

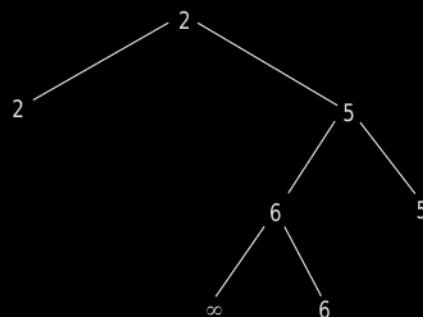
Considere la secuencia de entrada 2, 4, 7, 3, 1, 6, 5, 8. Es posible construir un árbol de knockout sort para encontrar el segundo número más pequeño, como se muestra en la figura 2-8.

FIGURA 2-8 Árbol de knockout sort para encontrar el número más pequeño.



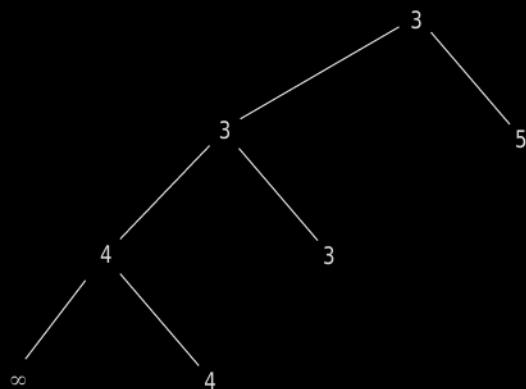
Una vez que se encuentra el número más pequeño, es posible comenzar a buscar el segundo número más pequeño al sustituir 1 por ∞ . Así, sólo es necesario analizar una pequeña porción del árbol de knockout sort, como se muestra en la figura 2-9.

FIGURA 2-9 Determinación del segundo número más pequeño.



Cada vez que se encuentra un número más pequeño, se sustituye por ∞ y así es más fácil encontrar el siguiente número más pequeño. Por ejemplo, ahora ya se han encontrado los dos primeros números más pequeños. Luego, el tercer número más pequeño puede encontrarse como se muestra en la figura 2-10.

FIGURA 2-10 Determinación del tercer número más pequeño con ordenamiento por knockout sort.



Para encontrar la complejidad temporal del ordenamiento por knockout sort:

El primer número más pequeño se encuentra después de $(n - 1)$ comparaciones. Para todas las demás selecciones, sólo se requieren $\lceil \log n \rceil - 1$ comparaciones. En consecuencia, el número total de comparaciones es

$$(n - 1) + (n - 1)(\lceil \log n \rceil - 1).$$

Así, la complejidad temporal del ordenamiento por knockout sort es $O(n \log n)$, que es igual a la cota inferior que se encontró en el apartado 2-4. El ordenamiento por knockout sort es, por consiguiente, un algoritmo de ordenamiento óptimo. Debe observarse que la complejidad temporal $O(n \log n)$ es válida para los casos mejor, promedio y peor.

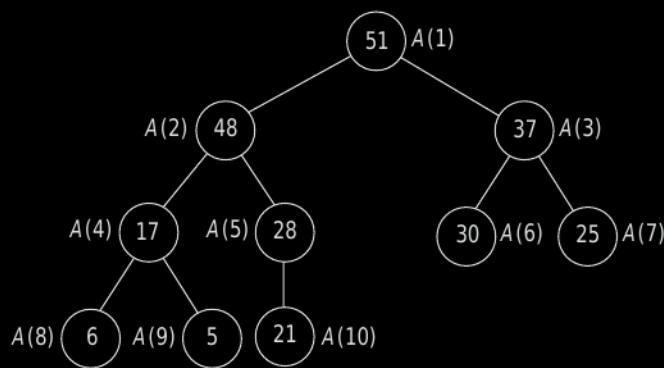
El ordenamiento por knockout sort es mejor que el ordenamiento por inserción directa porque usa información previa. Desafortunadamente, el árbol de knockout sort requiere espacio adicional. Para implementar el ordenamiento por knockout sort se requieren aproximadamente $2n$ posiciones. Este ordenamiento puede mejorarse por el ordenamiento heap sort, el cual se abordará en el resto de esta sección.

De manera semejante al ordenamiento por knockout sort, para almacenar los datos en el ordenamiento heap sort se utiliza una estructura de datos especial. Esta estructura se denomina heap. Un heap es un árbol binario que cumple las siguientes condiciones:

1. El árbol está completamente balanceado.
2. Si la altura del árbol binario es h , entonces las hojas pueden estar al nivel h o al nivel $h - 1$.
3. Todas las hojas al nivel h están a la izquierda tanto como sea posible.
4. Los datos asociados con todos los descendientes de un nodo son menores que el dato asociado con este nodo.

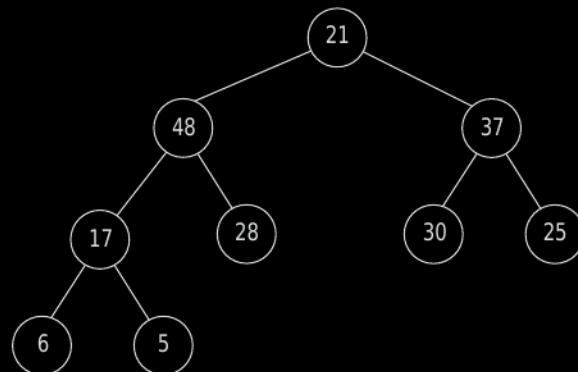
En la figura 2-11 se muestra un heap para 10 números.

FIGURA 2-11 Un heap.



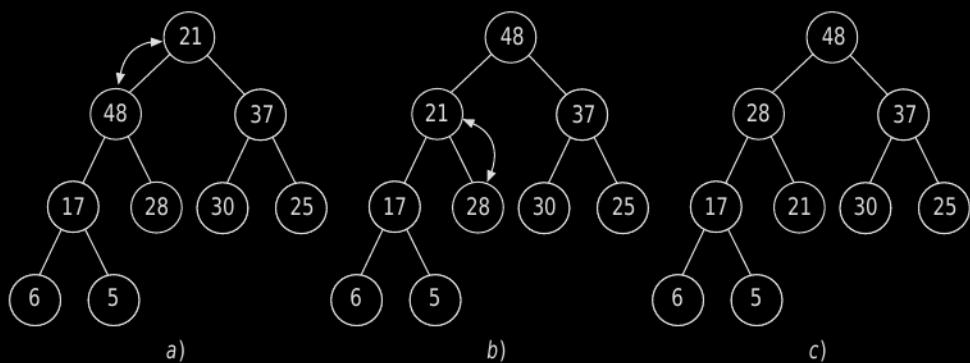
Por definición, la raíz del heap, $A(1)$, es el número más grande. Se supondrá que el heap ya está construido (la construcción de un heap se abordará más tarde). Así, es posible partir de $A(1)$. Luego de $A(1)$, que es el número más grande, el primer heap deja de ser un heap. Entonces, $A(1)$ se sustituye por $A(n) = A(10)$. Así, se tiene el árbol que se muestra en la figura 2-12.

FIGURA 2-12 Sustitución de $A(1)$ por $A(10)$.



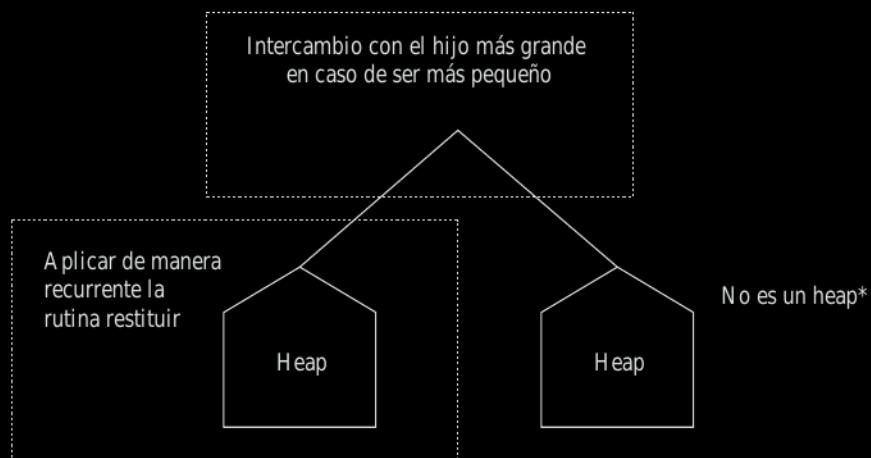
El árbol binario balanceado de la figura 2-12 no es un heap; sin embargo, es posible restituirlo fácilmente, como se muestra en la figura 2-13. El árbol binario de la figura 2-13c) es un heap.

FIGURA 2-13 Restitución de un heap.



La rutina restituir puede entenderse mejor con la figura 2-14.

FIGURA 2-14 La rutina restituir.



* Hasta no restituirlo, no es un heap. (*N. del R.T.*)

Algoritmo 2-6 □ Restore(*i, j*)

Input: $A(i), A(i+1), \dots, A(j)$.

Output: $A(i), A(i+1), \dots, A(j)$ como un heap.

Si $A(i)$ no es una hoja y si un hijo de $A(i)$ contiene un elemento más grande que $A(i)$, entonces

Begin

Hacer que $A(h)$ sea el hijo $A(i)$ con el elemento más grande

Intercambiar $A(i)$ y $A(h)$

Restore(h, j) (*restitución de un heap*)

End

El parámetro j se usa para determinar si $A(i)$ es una hoja o no y si $A(i)$ tiene dos hijos. Si $i > j/2$, entonces $A(i)$ es una hoja y restituir (i, j) no requiere hacer nada en absoluto porque $A(i)$ ya es un heap de por sí.

Puede afirmarse que hay dos elementos importantes en el ordenamiento heap sort:

1. Construcción del heap.
2. Eliminación del número más grande y restitución del heap.

Suponiendo que el heap ya está construido, entonces el ordenamiento heap sort puede describirse como sigue:

Algoritmo 2-7 □ Ordenamiento heap sort

Input: $A(1), A(2), \dots, A(n)$ donde cada $A(i)$ es un nodo de un heap ya construido.

Output: La secuencia ordenada de las $A(i)$.

For $i := n$ down to 2 do (*ciclo decreciente*)

Begin

Output $A(1)$

$A(1) := A(i)$

Delete $A(i)$

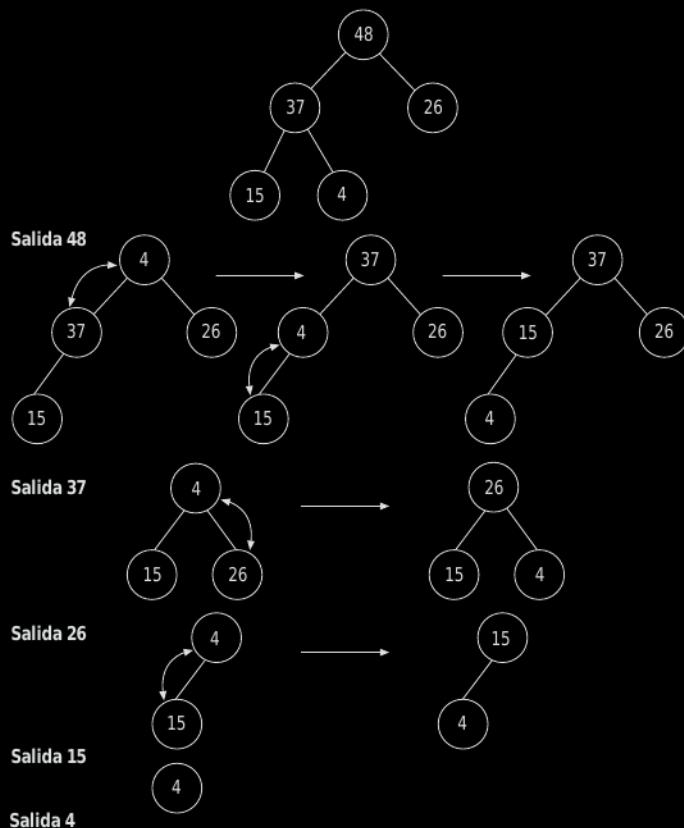
Restore ($1, i - 1$)

End

Output $A(1)$

► Ejemplo 2-6 Ordenamiento heap sort

Los pasos siguientes muestran un caso típico de ordenamiento heap sort.



Una bella característica de un heap es que es posible representarlo mediante un arreglo. Es decir, no se requieren apuntadores porque un heap es un árbol binario completamente balanceado. Cada nodo y sus descendientes pueden determinarse de manera única. La regla es más bien simple: los descendientes de $A(h)$ son $A(2h)$ y $A(2h + 1)$ en caso de existir.

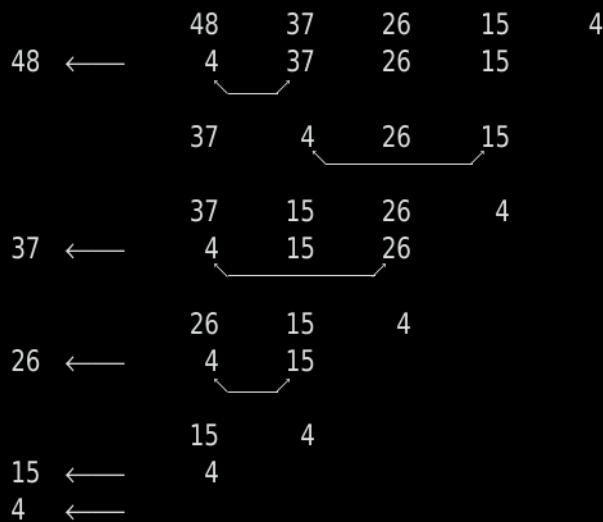
El heap de la figura 2-11 se almacena ahora como:

$A(1)$	$A(2)$	$A(3)$	$A(4)$	$A(5)$	$A(6)$	$A(7)$	$A(8)$	$A(9)$	$A(10)$
51	48	37	17	28	30	25	6	5	21

Considere, por ejemplo, $A(2)$. Su hijo izquierdo es $A(4) = 17$.

Considere $A(3)$. Su hijo derecho es $A(7) = 25$.

En consecuencia, todo el proceso de ordenamiento heap sort puede manejarse mediante una tabla o cuadro. Por ejemplo, el ordenamiento heap sort en el ejemplo 2-6 ahora puede describirse como sigue:



L a construcción de un heap

Para construir un heap considere la figura 2-14, donde el árbol binario no es un heap. Sin embargo, ambos subárboles debajo de la parte superior del árbol son heaps. Para esta clase de árboles es posible “construir” un heap usando la subrutina restitución. La construcción se basa en la idea anterior. Se empieza con cualquier árbol binario completamente balanceado arbitrario y gradualmente se transforma en un heap al invocar repetidamente la subrutina restitución.

Sea $A(1), A(2), \dots, A(n)$ un árbol binario completamente balanceado cuyos nodos hoja al nivel más elevado se encuentran lo más a la izquierda posible. Para este árbol binario, puede verse que $A(i), i = 1, 2, \dots, \lfloor n/2 \rfloor$ debe ser un nodo interno con descendientes y que $A(i), i = \lfloor n/2 \rfloor + 1, \dots, n$ debe ser un nodo hoja sin descendientes. Todos los nodos hoja pueden considerarse trivialmente como heaps. Así, no es necesario realizar ninguna operación sobre ellos. La construcción de un heap comienza desde la restitución del subárbol cuya raíz está en $\lfloor n/2 \rfloor$. El algoritmo para construir un heap es como sigue:

Algoritmo 2-8 □ Construcción de un heap

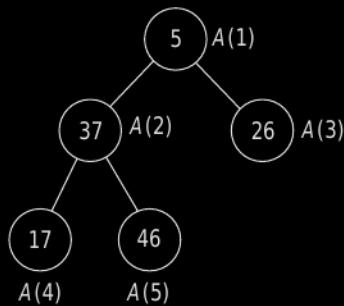
Input: $A(1), A(2), \dots, A(n)$.

Output: $A(1), A(2), \dots, A(n)$ como heap.

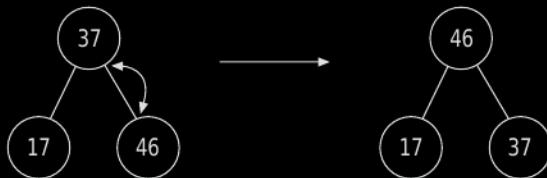
For $i := \lfloor n/2 \rfloor$ down to 1 do
 Restore (i, n)

► **Ejemplo 2-7 Construcción de un heap**

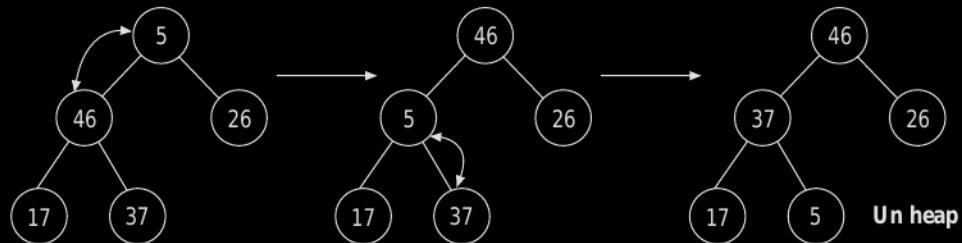
Considere el siguiente árbol binario que no es un heap.



En este heap, $n = 5$ y $\lfloor n/2 \rfloor = 2$. Por consiguiente, el subárbol cuya raíz está en $A(2)$ se restituye como sigue:



Luego se restituye $A(1)$:



El ordenamiento heap sort es una mejora del knockout sort porque para representar el heap se usa un arreglo lineal. A continuación se analiza la complejidad temporal del ordenamiento heap sort.

Análisis del peor caso de la construcción de un heap

Considere que hay n números para ordenar. La profundidad d de un árbol binario completamente balanceado es por lo tanto $\lfloor \log n \rfloor$. Para cada nodo interno es necesario

realizar dos comparaciones. Sea L el nivel de un nodo interno. Entonces, en el peor caso, para ejecutar la subrutina restituir es necesario efectuar $2(d - L)$ comparaciones. El número máximo de nodos en el nivel L es 2^L . Así, el número total de comparaciones para la etapa de construcción es a lo sumo

$$\sum_{L=0}^{d-1} 2(d - L)2^L = 2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L 2^{L-1}.$$

En el apartado 2-2, en la ecuación (2-1) se demostró que

$$\sum_{L=0}^k L 2^{L-1} = 2^k(k - 1) + 1.$$

En consecuencia,

$$\begin{aligned} \sum_{L=0}^{d-1} 2(d - L)2^L &= 2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L 2^{L-1} \\ &= 2d(2^d - 1) - 4(2^{d-1}(d - 1 - 1) + 1) \\ &= 2d(2^d - 1) - 4(d2^{d-1} - 2^d + 1) \\ &= 4 \cdot 2^d - 2^d - 4 \\ &= 4 \cdot 2^{\lfloor \log n \rfloor} - \lfloor 2 \log n \rfloor - 4 \\ &= cn - \lfloor 2 \log n \rfloor - 4 \quad \text{donde } 2 \leq c \leq 4 \\ &\leq cn. \end{aligned}$$

A sí, el número total de comparaciones necesarias para construir un heap en el peor caso es $O(n)$.

Complejidad temporal de eliminar elementos de un heap

Como se demostró, después de que se construye un heap, la parte superior de éste es el número más grande y ahora ya es posible eliminarlo (o sacarlo). A continuación se analiza el número de comparaciones necesarias para sacar todos los elementos numéricos de un heap que consta de n elementos. Observe que después de eliminar un número, en el peor caso, para restituir el heap se requieren $2\lfloor \log i \rfloor$ comparaciones si quedan i elementos. Así, el número total de pasos necesarios para eliminar todos los números es

$$2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor.$$

Para evaluar esta fórmula, se considerará el caso de $n = 10$.

$$\begin{aligned}\lfloor \log 1 \rfloor &= 0 \\ \lfloor \log 2 \rfloor &= \lfloor \log 3 \rfloor = 1 \\ \lfloor \log 4 \rfloor &= \lfloor \log 5 \rfloor = \lfloor \log 6 \rfloor = \lfloor \log 7 \rfloor = 2 \\ \lfloor \log 8 \rfloor &= \lfloor \log 9 \rfloor = 3.\end{aligned}$$

Se observa que hay

2^1 números iguales a $\lfloor \log 2^1 \rfloor = 1$

2^2 números iguales a $\lfloor \log 2^2 \rfloor = 2$

y $10 - 2^{\lfloor \log 10 \rfloor} = 10 - 2^3 = 2$ números iguales a $\lfloor \log n \rfloor$.

En general,

$$\begin{aligned}2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor &= 2 \sum_{i=1}^{\lfloor \log n \rfloor - 1} i 2^i + 2(n - 2^{\lfloor \log n \rfloor}) \lfloor \log n \rfloor \\ &= 4 \sum_{i=1}^{\lfloor \log n \rfloor - 1} i 2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor}) \lfloor \log n \rfloor.\end{aligned}$$

Al usar $\sum_{i=1}^k i 2^{i-1} = 2^k(k-1)+1$ (ecuación 2-1 en el apartado 2-2)

se tiene

$$\begin{aligned}2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor &= 4 \sum_{i=1}^{\lfloor \log n \rfloor - 1} i 2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor}) \lfloor \log n \rfloor \\ &= 4(2^{\lfloor \log n \rfloor - 1}(\lfloor \log n \rfloor - 1 - 1) + 1) + 2n \lfloor \log n \rfloor - 2 \lfloor \log n \rfloor 2^{\lfloor \log n \rfloor} \\ &= 2 \cdot 2^{\lfloor \log n \rfloor} \lfloor \log n \rfloor - 8 \cdot 2^{\lfloor \log n \rfloor - 1} + 4 + 2n \lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor} \lfloor \log n \rfloor \\ &= 2 \cdot n \lfloor \log n \rfloor - 4 \cdot 2^{\lfloor \log n \rfloor} + 4 \\ &= 2n \lfloor \log n \rfloor - 4cn + 4 \quad \text{donde } 2 \leq c \leq 4 \\ &= O(n \log n).\end{aligned}$$

En consecuencia, la complejidad temporal del peor caso para obtener todos los elementos de un heap en orden clasificado es $O(n \log n)$.

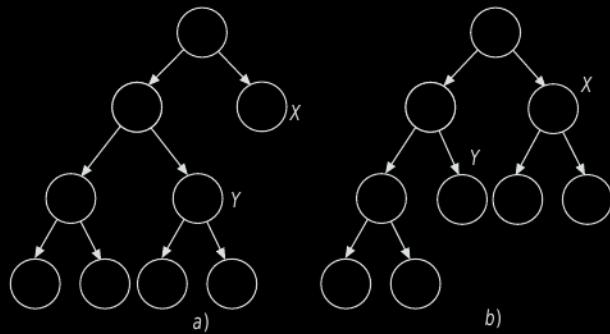
En resumen, se concluye que la complejidad temporal del peor caso del ordenamiento heap sort es $O(n \log n)$. Aquí se recalca que el ordenamiento heap sort alcanza esta complejidad temporal de $O(n \log n)$ esencialmente porque utiliza una estructura de datos de modo que cada operación de salida requiere a lo sumo $\lfloor \log i \rfloor$ pasos, donde i es el número de elementos restantes. Este inteligente diseño de estructura de datos es fundamental para el ordenamiento heap sort.

2-6 LA COTA INFERIOR DEL CASO PROMEDIO DEL ORDENAMIENTO

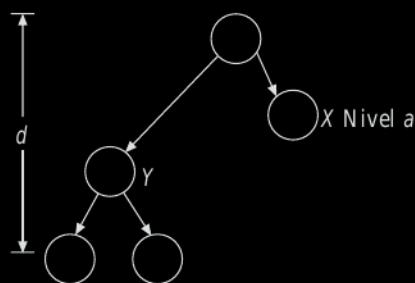
En el apartado 2-4 se estudió la cota inferior del peor caso del ordenamiento. En esta sección se deducirá la cota inferior del caso promedio del problema de ordenamiento. Seguirá usándose el modelo del árbol de decisión binario.

Como ya se analizó, todo algoritmo de ordenamiento basado en comparaciones puede describirse mediante un árbol de decisión binario. En este árbol la ruta que va de su raíz a un nodo hoja corresponde a la acción del algoritmo en respuesta a un caso particular de entrada. A demás, la longitud de esta ruta es igual al número de comparaciones ejecutadas para este conjunto de datos de entrada. Definiremos *la longitud de la ruta externa de un árbol como la suma de las longitudes de las rutas que van de la raíz a cada uno de los nodos hoja*. Así, la complejidad temporal media de un algoritmo de ordenamiento basado en comparaciones es igual a la longitud de la ruta externa del árbol de decisión binario correspondiente a este algoritmo dividida entre el número de nodos hoja, que es $n!$

Para encontrar la cota inferior de la complejidad temporal del ordenamiento es necesario determinar la mínima longitud de la ruta externa de todos los árboles binarios posibles con $n!$ nodos hoja. Entre todos los árboles binarios posibles con un número fijo de nodos hoja, la mínima longitud de la ruta externa se minimiza si el árbol está balanceado. Es decir, todos los nodos hoja están en el nivel d o en el nivel $d - 1$ para alguna d . Considere la figura 2-15. En la figura 2-15a), el árbol no está balanceado. La longitud de la ruta externa de este árbol es $4 \times 3 + 1 = 13$. A hora ya es posible reducir esta longitud de la ruta externa eliminando los dos descendientes de Y y asignándolos a X . Así, la longitud de la ruta externa se convierte ahora en $2 \times 3 + 3 \times 2 = 12$.

FIGURA 2-15 Modificación de un árbol binario no balanceado.

El caso general se describe ahora en la figura 2-16. En esta figura, suponga que en el nivel a hay un nodo hoja y que la profundidad del árbol es d , donde $a \leq d - 2$. Este árbol puede modificarse de modo que la longitud de la ruta externa se reduzca sin cambiar el número de nodos hoja. Para modificar el árbol, se selecciona cualquier nodo en el nivel $d - 1$ que tenga descendientes en el nivel d . El nodo hoja que está en el nivel a y el nodo que está en el nivel $d - 1$ se denotan por X y Y , respectivamente. Se quitan los descendientes de Y y se asignan a X . Para el nodo Y , éste originalmente tenía dos descendientes y la suma de la longitud de sus rutas es $2d$. Luego, Y se vuelve un nodo hoja y la longitud de su ruta es $d - 1$. Esta eliminación reduce la longitud de la ruta externa por $2d - (d - 1) = d + 1$. Para X , éste originalmente era un nodo hoja. A hora se convierte en un nodo interno y sus dos nodos descendientes se vuelven nodos hoja. En un principio, la longitud de la ruta era a . A hora la suma de las dos longitudes de las rutas es $2(a + 1)$. Así, esta colocación incrementa la longitud de la ruta externa por $2(a + 1) - a = a + 2$. El cambio neto es $(d + 1) - (a + 2) = (d - a) - 1 \geq 2 - 1 = 1$. Es decir, el cambio neto es disminuir la longitud de la ruta externa.

FIGURA 2-16 Árbol binario no balanceado.

En consecuencia, se concluye que un árbol binario no balanceado puede modificarse de modo que la longitud de la ruta externa disminuya y la longitud de la ruta externa de un árbol binario se minimiza si y sólo si el árbol está balanceado.

Considere que en total hay x nodos hoja. A hora se calculará la longitud de la ruta externa de un árbol binario balanceado que tiene c nodos hoja. Esta longitud de la ruta externa se encuentra aplicando el siguiente razonamiento:

1. La profundidad del árbol es $d = \lceil \log c \rceil$. Los nodos hoja sólo pueden aparecer en el nivel d o en el nivel $d - 1$.
2. En el nivel $d - 1$ hay x_1 nodos hoja y en el nivel d hay x_2 nodos hoja. Luego,

$$x_1 + x_2 = c.$$

3. Para simplificar el análisis se supondrá que el número de nodos en el nivel d es par. El lector podrá observar fácilmente que si el número de nodos en el nivel d es impar, entonces el siguiente resultado sigue siendo verdadero. Para cada dos nodos en el nivel d , en el nivel $d - 1$ hay un nodo padre. Este nodo padre no es un nodo hoja. Así, se tiene la siguiente ecuación

$$x_1 + \frac{x_2}{2} = 2^{d-1}.$$

4. Al resolver estas ecuaciones se obtiene

$$\frac{x_2}{2} = c - 2^{d-1}$$

$$x_2 = 2(c - 2^{d-1})$$

$$x_1 = 2^d - c.$$

5. La longitud de la ruta externa es

$$\begin{aligned} & x_1(d - 1) + x_2 d \\ &= (2^d - c)(d - 1) + (2c - 2^d)d \\ &= c + cd - 2^d. \end{aligned}$$

6. Debido a que $d = \lceil \log c \rceil$, al sustituir $\log c \leq d < \log c + 1$ en la ecuación anterior, se tiene $c + cd - 2^d \geq c + c(\log c) - 2 \cdot 2^{\log c} = c \log c - c$. Así, la longitud de la ruta externa es mayor que $c \log c - c = n! \log n! - n!$ En consecuencia, la complejidad temporal del ordenamiento en el caso promedio es mayor que

$$\frac{n! \log n! - n!}{n!} = \log n! - 1.$$

Al usar el resultado que se analizó en el apartado 2-4, ahora se concluye que *la cota inferior en el caso promedio del problema de ordenamiento es $\Omega(n \log n)$* .

En el ejemplo 2-4 del apartado 2-2 se demostró que la complejidad temporal en el caso promedio del quick sort es $O(n \log n)$. Así, el quick sort es óptimo por lo que se refiere a su desempeño en el caso promedio.

En el ejemplo 2-3 del apartado 2-2 se demostró que la complejidad temporal en el caso promedio del ordenamiento por selección directa también es $O(n \log n)$. Sin embargo, debe entenderse que esta complejidad temporal es en términos del cambio de señal. El número de comparaciones para el ordenamiento por selección directa es $n(n - 1)/2$ en los casos promedio y peor. Debido a que el número de comparaciones es un factor temporal dominante en la programación práctica, resulta que en la práctica el ordenamiento por selección directa es bastante lento.

La complejidad temporal en el caso promedio del famoso ordenamiento por burbuja, así como el ordenamiento por inserción directa, es $O(n^2)$. La experiencia indica que el ordenamiento por burbuja es mucho más lento que el quick sort.

Finalmente se abordará el ordenamiento heap sort que se analizó en el apartado 2-5. La complejidad temporal en el peor caso del ordenamiento heap sort es $O(n \log n)$ y la complejidad temporal en el caso promedio del ordenamiento heap sort nunca ha sido determinada. Sin embargo, se sabe que debe ser mayor o igual a $O(n \log n)$ debido a la cota inferior que se encontró en esa sección. Pero no puede ser mayor que $O(n \log n)$ porque su complejidad temporal en el peor caso es $O(n \log n)$. En consecuencia, es posible deducir el hecho de que la complejidad temporal en el caso promedio del ordenamiento heap sort es $O(n \log n)$.

2-7 / CÓMO MEJORAR UNA COTA INFERIOR MEDIANTE ORÁCULOS

En la sección previa se demostró cómo usar el modelo del árbol de decisión binario a fin de obtener una cota inferior para el ordenamiento. Es simplemente afortunado que la cota inferior haya sido tan buena. Es decir, existe un algoritmo cuya complejidad temporal en el peor caso es exactamente igual a esta cota inferior. En consecuencia, puede tenerse la certeza de que ya no es posible hacer más alta esta cota inferior.

En esta sección se presentará un caso en que el modelo del árbol de decisión binario no produce una cota inferior muy significativa. Es decir, se mostrará que aún es posible mejorar la cota inferior obtenida usando el modelo del árbol de decisión binario.

Considere el problema de fusión. Si el algoritmo de fusión (merge) se basa en la operación de comparación e intercambio, entonces es posible usar el modelo del árbol de decisión. Se puede deducir una cota inferior de fusión aplicando el razonamiento para la obtención de la cota inferior del ordenamiento. En el ordenamiento, el número

de nodos hoja es $n!$, de modo que la cota inferior para el ordenamiento es $\lceil \log_2 n! \rceil$. En la fusión, el número de nodos hoja sigue siendo el número de casos distintos que quieren distinguirse. Así, dadas dos secuencias ordenadas A y B de m y n elementos, respectivamente, ¿cuántas secuencias fusionadas diferentes posibles hay? De nuevo, para simplificar el análisis, se supondrá que todos los $(m + n)$ elementos son distintos. Después de que n elementos se han fusionado en m elementos, en total hay $\binom{m+n}{n}$ formas de fusionarlos sin perturbar el orden original de las secuencias A y B . Esto significa que es posible obtener una cota inferior para la fusión como

$$\left\lceil \log \binom{m+n}{n} \right\rceil.$$

Sin embargo, jamás se ha determinado ningún algoritmo de fusión con el que se obtenga esta cota inferior.

A continuación se considerará un algoritmo de fusión convencional que compara los elementos superiores de dos listas ordenadas y da como salida el menor. Para este algoritmo de fusión, la complejidad temporal en el peor caso es $m + n - 1$, que es mayor que o igual a

$$\left\lceil \log \binom{m+n}{n} \right\rceil.$$

Es decir, se tiene la siguiente desigualdad:

$$\left\lceil \log \binom{m+n}{n} \right\rceil \leq m + n - 1.$$

¿Cómo puede establecerse un puente en la brecha? Según el análisis hecho, es posible ya sea incrementar la cota inferior o encontrar un mejor algoritmo cuya complejidad temporal sea más baja. De hecho, resulta interesante observar que cuando $m = n$, otra cota inferior de la fusión es $m + n - 1 = 2n - 1$.

Esto se demostrará mediante el enfoque del oráculo. Un oráculo proporcionará un caso muy difícil (un dato de entrada particular). Si se aplica cualquier algoritmo a este conjunto de datos, el algoritmo deberá trabajar bastante para resolver el problema. Al usar este conjunto de datos, es posible deducir una cota inferior para el peor caso.

Suponga que se tienen dos secuencias a_1, a_2, \dots, a_n y b_1, b_2, \dots, b_n . A demás, considere el muy difícil caso en que $a_1 < b_1 < a_2 \dots a_n < b_n$. Suponga que algún algoritmo de fusión ya ha fusionado correctamente a_1, a_2, \dots, a_{i-1} con b_1, b_2, \dots, b_{i-1} y que produce la siguiente secuencia ordenada:

$$a_1, b_1, \dots, a_{i-1}, b_{i-1}.$$

No obstante, suponga que este algoritmo de fusión no compara a_i con b_i . Resulta evidente que no hay ninguna forma en que el algoritmo haga una decisión correcta sobre si a_i , o b_i , debe colocarse al lado de b_{i-1} . Así, es necesario comparar a_i y b_i . APLICANDO un razonamiento semejante puede demostrarse que es necesario comparar b_i y a_{i+1} después que a_i se ha escrito al lado de b_{i-1} . En resumen, cada b_i debe compararse con a_i y a_{i+1} . En consecuencia, cuando $m = n$, cualquier algoritmo de fusión requiere efectuar en total $2n - 1$ comparaciones. Quisiéramos recordar al lector que esta cota inferior $2n - 1$ para fusionar sólo es válida para el caso en que $m = n$.

Debido a que el algoritmo de fusión convencional requiere $2n - 1$ comparaciones para el caso en que $m = n$, puede concluirse que el algoritmo de fusión convencional es óptimo porque su complejidad temporal en el peor caso es igual a esta cota inferior.

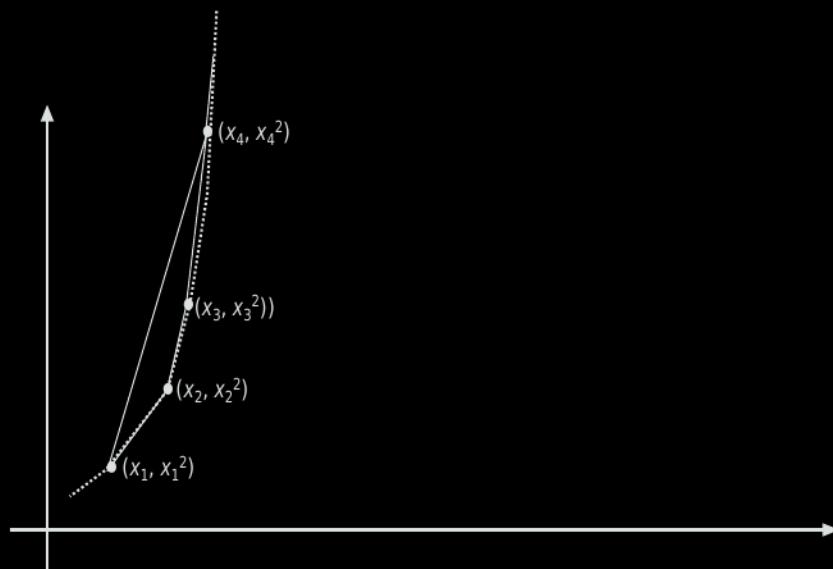
El análisis anterior muestra que algunas veces es posible mejorar una cota inferior con una más alta.

2-8 / DETERMINACIÓN DE LA COTA INFERIOR POR TRANSFORMACIÓN DEL PROBLEMA

En la sección previa se encontraron cotas inferiores mediante el análisis directo de los problemas. A algunas veces esto parece difícil. Por ejemplo, el problema de la cubierta convexa (convex hull) consiste en encontrar el menor polígono (cubierta) convexo de un conjunto de puntos en el plano. ¿Cuál es la cota inferior del problema de la cubierta convexa? Parece más bien difícil encontrar directamente una cota inferior para este problema. No obstante, a continuación se demostrará que es fácil obtener una cota inferior bastante significativa mediante la transformación del problema de ordenamiento, cuya cota inferior se conoce, para este problema.

Sea x_1, x_2, \dots, x_n el conjunto de puntos a clasificar y, sin perder la generalidad, puede suponerse que $x_1 < x_2 < \dots < x_n$. Luego cada x_i se asocia con x_i^2 a fin de obtener un punto bidimensional (x_i, x_i^2) . Todos estos puntos recién creados están en la parábola $y = x^2$. Considere la cubierta convexa construida con estos $n(x_i, x_i^2)$ puntos. Como se muestra en la figura 2-17, esta cubierta convexa consta de una lista de números clasificados. En otras palabras, al resolver el problema de la cubierta convexa también es posible resolver el problema de ordenamiento. El tiempo total del ordenamiento es igual al tiempo necesario para la transformación más el tiempo necesario para resolver el problema de la cubierta convexa. Así, la cota inferior del problema de la cubierta convexa es igual a la cota inferior del problema de ordenamiento menos el tiempo necesario para la transformación. Es decir, la cota inferior del problema de la cubierta convexa no es menor que $\Omega(n \log n) - \Omega(n) = \Omega(n \log n)$ cuando la transformación requiere $\Omega(n)$ pasos. El lector observará que esta cota inferior no puede hacerse más alta porque hay un algoritmo para resolver el problema de la cubierta convexa en $\Omega(n \log n)$ pasos.

FIGURA 2-17 Cubierta convexa construida a partir de los datos de un problema de ordenamiento.



En general, suponga que se quiere encontrar una cota inferior para el problema P_1 . Sea P_2 un problema cuya cota inferior se desconoce. A demás, suponga que P_2 puede transformarse en P_1 , de modo que P_2 puede resolverse después que se ha resuelto P_1 . Sean $\Omega(f_1(n))$ y $\Omega(f_2(n))$ que denotan las cotas inferiores de P_1 y P_2 , respectivamente. Sea $O(g(n))$ el tiempo necesario para transformar P_2 en P_1 . Entonces

$$\begin{aligned}\Omega(f_1(n)) + O(g(n)) &\geq \Omega(f_2(n)) \\ \Omega(f_1(n)) &\geq \Omega(f_2(n)) - O(g(n)).\end{aligned}$$

A continuación se proporciona otro ejemplo para demostrar la factibilidad de este enfoque. Suponga que se quiere encontrar la cota inferior del problema del árbol de expansión euclíadiano mínimo. Debido a que es difícil obtener directamente la cota inferior de P_1 , se considera P_2 , que nuevamente es el problema de ordenamiento. Entonces se define la transformación: para todo x_i , sea $(x_i, 0)$ un punto bidimensional. Puede verse que el ordenamiento se completará tan pronto como se construya el árbol de expansión mínimo a partir de los $(x_i, 0)$. Nuevamente se supondrá que $x_1 < x_2 < \dots < x_n$. Entonces en el árbol de expansión mínimo hay un borde entre $(x_i, 0)$ y $(x_j, 0)$ si y sólo si $j = i + 1$. En consecuencia, la solución del problema del árbol de expansión mínimo euclíadiano también es una solución del problema de ordenamiento. De nuevo se observa que una cota inferior significativa del árbol de expansión mínimo euclíadiano es $\Omega(n \log n)$.

2-9 NOTAS Y REFERENCIAS

En este capítulo se presentan algunos conceptos básicos del análisis de algoritmos. Para profundizar en el estudio del tema de análisis de algoritmos, consulte los siguientes autores: Basse y Van Gelder (2000); Aho, Hopcroft y Ullman (1974); Greene y Knuth (1981); Horowitz, Sahni y Rajasekaran (1998) y Purdom y Brown (1985a). Varios ganadores del premio Turing son excelentes investigadores de algoritmos. En 1987, la ACM Press publicó una colección de conferencias de 20 ganadores del premio Turing (Ashenhurst, 1987). En este volumen, todas las conferencias de Rabin, Knuth, Cook y Karp se refieren a las complejidades de los algoritmos. El premio Turing de 1986 fue otorgado a Hopcroft y Tarjan. La conferencia de Tarjan durante la recepción del premio Turing trataba del diseño de algoritmos y puede consultarse en Tarjan (1987). Weide (1977) también aportó una investigación sobre técnicas de análisis de algoritmos.

En este capítulo se presentaron varios algoritmos de ordenamiento. Para conocer un análisis más detallado sobre ordenamiento y búsqueda, puede consultarse la obra de Knuth (1973). Para más material sobre el análisis del ordenamiento por inserción directa, la búsqueda binaria y el ordenamiento por selección directa, consulte las secciones 5.2.1, 6.2.1 y la sección 5.2.3, respectivamente, de la obra de Knuth (1973). El quick sort fue obra de Hoare (1961, 1962). El ordenamiento heap sort fue descubierto por Williams (1964). Para más detalles sobre la determinación de la cota inferior de un ordenamiento, consulte la sección 5.3.1 de la obra de Knuth (1973); acerca del ordenamiento knockout sort consulte la sección 5.2.3 del mismo autor (1973).

La terminología básica de un árbol puede encontrarse en muchos libros de texto sobre estructura de datos. Por ejemplo, vea las secciones 5.1 y 5.2 de Horowitz y Sahni (1976). La profundidad de un árbol también se conoce como la altura de un árbol. Para saber más sobre el análisis de la longitud de la ruta externa y el efecto del árbol binario completo, consulte la sección 2.3.4.5 del libro de Knuth (1969).

En la sección 6.1 de la obra de Preparata y Shamos (1985) puede encontrarse un estudio sobre el problema del árbol de expansión mínimo. Hay más información sobre el problema de la determinación del rango en la sección 4.1 de libro de Shamos (1978) y en la sección 8.8.3 del libro de Preparata y Shamos (1985). La demostración de que la complejidad temporal de tiempo medio para encontrar la mediana es $O(n)$ puede consultarse en la sección 3.6 de la obra de Horowitz y Sahni (1978).

Para consultar material sobre el mejoramiento de una cota inferior a través de oráculos, vea la sección 5.3.2 del libro de Knuth (1973) y también la sección 10.2 de la obra de Horowitz y Sahni (1978). En cuanto a material sobre la determinación de cotas inferiores mediante la transformación del problema, consulte las secciones 3.4 y 6.1.4 del libro de Shamos (1978) y también las secciones 3.2 y 5.3 del de Preparata y Shamos (1985). En Shamos (1978) y Preparata y Shamos (1985) hay muchos ejemplos que prueban las cotas inferiores por transformación.

2-10 BIBLIOGRAFÍA ADICIONAL

Las teorías sobre cotas inferiores siempre han atraído a los investigadores. Algunos artículos que se han publicado recientemente sobre este tema son de los siguientes autores: Dobkin y Lipton (1979); Edwards y Elphick (1983); Frederickson (1984); Fredman (1981); Grandjean (1988); Hasham y Sack (1987); Hayward (1987); John (1988); Karp (1972); McDiarmid (1988); McEhlhorn, Naher y Alt (1988); Moran, Snir y Manber (1985); Nakayama, Nishizeki y Saito (1985); Rangan (1983); Traub y Wozniakowski (1984); Yao (1981), y Yao (1985).

Para algunos artículos muy interesantes de reciente publicación, consulte Berman, Karpinski, Larmore, Plandowski y Rytter (2002); Blazewicz y Kasprzak (2003); Bodlaender, Downey, Fellows y Wareham (1995); Boldi y Vigna (1999); Bonizzoni y Vedova (2001); Bryant (1999); Cole (1994); Cole y Hariharan (1997); Cole, Farach-Colton, Hariharan, Przytycka y Thorup (2000); Cole, Hariharan, Paterson y Zwick (1995); Crescenzi, Goldman, Papadimitriou, Piccolboni y Yannakakis (1998); Darve (2000); Day (1987); Decatur, Goldreich y Ron (1999); Demri y Schnoebelen (2002); Downey, Fellows, Vardy y Whittle (1999); Hasewaga y Horai (1991); Hoang y Thierauf (2003); Jerrum (1985); Juedes y Lutz (1995); Kannan, Lawler y Warnow (1996); Kaplan y Shamir (1994); Kontogiannis (2002); Leoncini, Manzini y Margara (1999); Maes (1990); Mäier (1978); Marion (2003); Martinez y Roura (2001); Matousek (1991); Naor y Ruah (2001); Novak y Wozniakowski (2000); Owolabi y McGregor (1988); Pacholski, Szwast y Tendera (2000); Peleg y Rubinovich (2000), y Ponzio, Radhakrishnan y Venkatesh (2001).

Ejercicios

- 2.1 Proporcione los números de intercambios necesarios para los casos mejor, peor y promedio en el ordenamiento por burbuja, cuya definición puede encontrarse en casi todos los libros de texto sobre algoritmos. Los análisis de los casos mejor y peor son triviales. El análisis para el caso promedio puede realizarse mediante el siguiente proceso:
1. Defina la inversa de una permutación. Sea a_1, a_2, \dots, a_n una permutación del conjunto $(1, 2, \dots, n)$. Si $i < j$ y $a_j < a_i$, entonces (a_i, a_j) se

denomina inversión de esta permutación. Por ejemplo, $(3, 2)$ $(3, 1)$ $(2, 1)$ $(4, 1)$ son, todas, inversiones de la permutación $(3, 2, 4, 1)$.

2. Encuentre la relación entre la probabilidad de que una permutación dada tenga exactamente k inversiones y la probabilidad de que el número de permutaciones de n elementos tenga exactamente k inversiones.
 3. Aplique inducción para demostrar que el número medio de intercambios necesarios para el ordenamiento por burbuja es $n(n - 1)/4$.
- 2.2 Escriba un programa para ordenamiento por burbuja. Realice un experimento para convencerse de que, en efecto, el desempeño medio del algoritmo es $O(n^2)$.
- 2.3 Encuentre el algoritmo del algoritmo de Ford-Johnson para ordenar, que aparece en muchos libros de texto sobre algoritmos. Se demostró que este algoritmo es óptimo para $n \leq 12$. Implemente este algoritmo en una computadora y compárela con cualquier otro algoritmo de ordenamiento. ¿Le agrada este algoritmo? En caso negativo, intente determinar qué falla en el análisis.
- 2.4 Demuestre que para clasificar cinco números se requieren por lo menos siete comparaciones. Luego, demuestre que el algoritmo de Ford-Johnson alcanza esta cota inferior.
- 2.5 Demuestre que para encontrar el número más grande en una lista de n números, por lo menos se requieren $n - 1$ comparaciones.
- 2.6 Demuestre que para encontrar el segundo elemento más grande de una lista de n números por lo menos se requieren $n - 2 + \lceil \log n \rceil$ comparaciones.

Sugerencia: No es posible determinar el segundo elemento más grande sin haber encontrado el primero. Así, el análisis puede efectuarse como sigue:

1. Demuestre que para encontrar el elemento más grande se requieren por lo menos $n - 1$ comparaciones.
2. Demuestre que siempre hay alguna secuencia de comparaciones que obliga a encontrar al segundo elemento más grande en $\lceil \log n \rceil - 1$ comparaciones adicionales.

2.7 Demuestre que si $T(n) = aT\left(\frac{n}{b}\right) + n^c$, entonces para n una potencia de b y

$$T(1) = k, T(n) = ka^{\log_b n} + n^c \left(\frac{b}{a - b^c} \right) \left(\left(\frac{a}{b^c} \right)^{\log_b n} - 1 \right).$$

2.8 Demuestre que si $T(n) = \sqrt{n}T(\sqrt{n}) + n$, $T(m) = k$ y $m = n^{1/2^j}$, entonces

$$T(n) = kn^{(2^j-1)/2^j} + in.$$

2.9 Lea el teorema 10.5 que aparece en la obra de Horowitz y Sahni (1978). La demostración de este teorema constituye un buen método para encontrar una cota inferior.

2.10 Demuestre que la búsqueda binaria es óptima para todo algoritmo de búsqueda que sólo realice comparaciones.

2.11 Dados los siguientes pares de funciones, ¿cuál es el menor valor de n de modo que la primera función sea mayor que la segunda?

- a) $2^n, 2n^2$.
- b) $n^{1.5}, 2n \log_2 n$.
- c) $n^3, 5n^{2.81}$.

2.12 ¿El tiempo $\Omega(n \log n)$ es una cota inferior para el problema de clasificar n enteros que varían de 1 a C , donde C es una constante? ¿Por qué?